Dynamic Difficulty Adjustment via Procedural Level Generation Guided by a Markov Decision Process

A Dissertation Presented
by

Colan F. Biemer

to

Khoury College of Computer Sciences

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Computer Science

Northeastern University
Boston, Massachusetts

September 2025

Contents

List of Figures

List of Tables

Acknowledgments

I will admit that part of me wanted to do something silly like change what Connor McGregor said, “I would like to apologize... to absolutely nobody!” to, “I would like to thank... absolutely nobody!” That certainly would have saved me the time I’ve spent writing everything below, but I do have a lot of people I would like to thank, and it would feel wrong not to do so. So, here we go.

To begin with, I would like to thank my advisor, Seth Cooper. You’ve been an incredible advisor. You’ve helped me a lot over the years with my research and shown me a better way to approach problem-solving, teaching, and more. Thank you.[1] Also, I’d like to thank the Northeast Big Data Innovation Hub Seed Fund for funding that work.

At the beginning of this process, I was co-advised by Seth and Magy Seif El-Nasr. Magy, you helped me a great deal at the start of this process, and I learned a ton from working with you and the team on OPM. I would also like to thank the National Science Foundation for funding me while I had the privilege to work on OPM.

There are also many lab members to whom I owe my gratitude. To keep this short, I’ve gone with a list ordered in terms of “about” when we met: Sophie Spatharioti, Josh Miller, Chaima Jemmali, Nathan Partlan, Anurag Sarkar, Abdelrahman Madkour, Zhaoqing Teng, Erica Kleinman, Kutub Gandhi, Mahsa Bazzaz, Fiona Shyne, and Kaylah Facey. All of you have made my life at Northeastern better and helped me through this crazy process. Thank you.

Without a committee, there can be no dissertation. So, in alphabetical order, I would like to thank my committee members: Robert Platt, Bob De Schutter, and Robert Zubek. Based on the length of this dissertation alone, I think it is fair to say that it takes a kind and generous person to be willing to be on a committee, let alone on a committee with someone you’ve never met in person. Thank you.

In addition to Seth, some teachers and professors along the way have been very influential in my life. (I’m referring to the ones who have been a positive influence, of course.) For this paragraph, I’m ordering in terms of recency. First on the list is Mike Shah. Mike, you were great to work with, and I learned a ton from you. The way you approach teaching is, I think, the right way. Thank you. Frank Lee, you gave me a chance back when I was a sophomore in college at Drexel with an unimpressive resume to work on a project that changed my life. You’ve also been a great mentor and friend. Thank you. Roderick Mobley, you caught me at an interesting time when I was a high schooler in your chemistry class, but I think you handled my, let’s say, enthusiasm with grace and kindness, for which I am grateful. Thank you. Lastly, Kevin Randolph, you helped me get to a

[1] If any prospective students are out there and have, for some reason, found themselves reading this dissertation, I heartily recommend working with Seth. You won’t regret it.

place where I began to take school more seriously. I've also learned that you secretly gave me a test with a larger font, resulting in better grades for me. Thank you.

Outside of academia, I owe many people my thanks. Four of my friends, listed in alphabetical order by last name, stand out: Jake Bruce, Patrick McClanahan, Tyler Salathe, and Patrick White. All of you have made my life infinitely better, and I would have probably gone insane during the pandemic if not for Friday Night Games. Thank you.

One thank you that I have seen in almost every acknowledgments section is directed towards one's family. I'm lucky enough to be someone who has the privilege not only to have a family to thank but also to want to thank them. I won't list every name here, as there are too many to include without risking missing someone. However, all of you are incredible people and have been wonderfully kind to me over the years. Thank you.

I do feel the need to give a special shout-out to Garrett (my brother) and Rachel (my sister-in-law). You two are the best, and I am outrageously lucky to have you both in my life. Thank you. Thanks to you two, I have also become a funcle, which has been a joy and a privilege for me. Jaimie, Evan, and Cora, all three of you are the best. You have even playtested some of the games in this dissertation and helped me brainstorm mechanics and titles. Thank you.

Now for something extra trite: Last but not least, my parents. It's useless to say that I wouldn't be here without you because I literally would not be here without you. But that useless, meaningless saying that I genuinely dislike (if you couldn't tell) is important in times like this because it conveys a simple fact: you two were and are essential. I'm very lucky to have you both as my parents. Thank you.

Abstract of the Dissertation

Dynamic Difficulty Adjustment via Procedural Level Generation Guided by a Markov Decision Process

by

Colan F. Biemer

Doctor of Philosophy in Computer Science

Northeastern University, September 2025

Seth Cooper, Adviser

Procedural level generation (PLG) can create novel levels and improve the replayability of games, but there are requirements for a generated level. First, a level must be completable. Second, a level must look and feel like a level that would exist in the game, meaning more than a random combination of level components that happens to be completable is required. On top of these two requirements, though, is the player experience. If a level is too hard, the player will be frustrated. If too easy, they will be bored. Neither outcome is desirable. A PLG system should take the player into account. A Markov Decision Process can assemble levels tailored to a player while incorporating the designer's intent. We conducted user studies to evaluate the approach, and it was effective for both computational agents and human players on a platformer and a roguelike.

Part I

Introduction

Chapter 1

Introduction

"Bah," said Scrooge, "Humbug."

Charles Dickens, *A Christmas Carol*

If you have played video games, you've likely had the experience of being bored. The levels are too easy.[1] There isn't enough challenge. Similarly, you may have played a game that felt too difficult. Maybe an enemy felt unkillable. Maybe the level had a jumping puzzle with a platform that was too small. Maybe a puzzle seemed unsolvable. The list of why a level can be too easy or too hard is long, perhaps unending.

Video game designers are the ones who have to balance their game to make it neither too easy nor too hard. This is, of course, a very challenging task. Imagine a designer is building a progression for a game where the progression is a series of levels; for example, *Super Mario Bros.* [131] is a game where the progression to beat the game is a series of levels, see Figure 1.1. In *Mario*, the player starts at "1-1" and can only go to the next level in the progression, "1-2", if they beat the former. Meaning, the player replays a level until they win.[2] This is called a Static Level Progression (SLP), which is a level progression that does not change.

When building a SLP, the designer's goal is not to make a series of levels that the player will always beat, nor is their goal to make a series of levels that the player will never beat. They want to create challenging levels that teach players how to play their game. If the challenge ramps up too quickly, they will lose players because the game is too hard. If the challenge does not ramp up

[1] A level is "an environment or location where game play occurs" [143].

[2] This isn't entirely true in the case of *Mario* because players have a set number of hearts that limits the number of times that a player can replay a level.

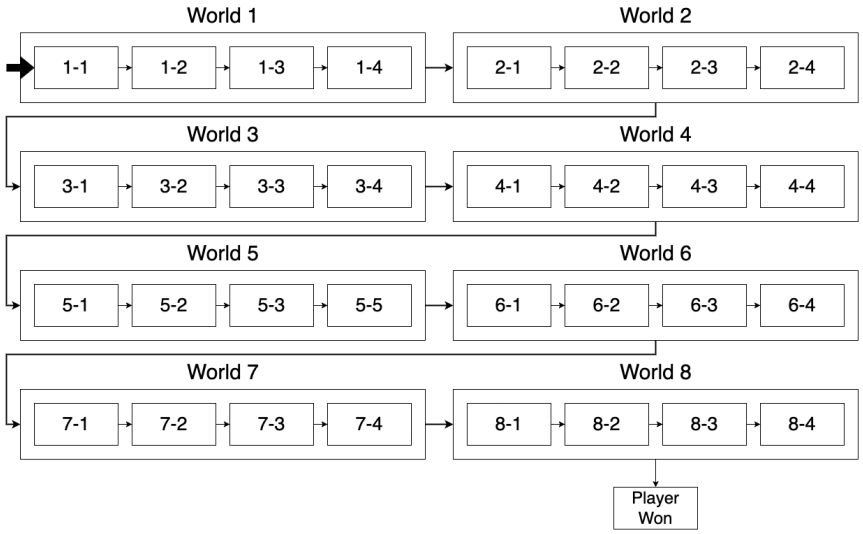


Figure 1.1: *Mario* has a static progression [131]. There are eight worlds, each with four levels. Players start the first level, *1-1*, with three lives. If they run out of lives, they have to start back at *1-1*, even if they made it to *8-4*. There are, however, ways for players to gain lives through the levels, which make the static progression less penalizing.

quickly enough, they will lose players because the game is too easy [42]. Making such a progression for one single player is already a challenging design task.

The designer, though, is not making a SLP for one specific player; they are trying to make a SLP that works for every player. This is an impossible task. Every player is different. Players start games with all kinds of backgrounds. Some will be highly skilled, while others will not. Some players may be playing a game for the first time, and others may have played hundreds of similar games. A SLP cannot accommodate all these players. If the SLP is built to favor teaching players new to games, high-skilled players will be bored. If the SLP is built to be extremely difficult, weaker players will be overwhelmed. As a result, designers are stuck in a *Catch-22* [79].

Despite all these problems, SLPs are still used, and there are three main reasons:

1. *Control* - A SLP lets the designer take full ownership over the player's experience. There are no unexpected outcomes, which is important when the game is a commercial product.

2. *Ease of Implementation* - SLPs are simple to implement. Given a linear progression like *Mario*, you can picture an array of strings, where each string links to a level file. Then the player's current progression is simply an index into that array. Even something more complex,

such as a progression with branching, is simply a tree of strings, and a reference to the current node the player is at.

3. *Ease of Modification* - The designer can easily modify the ordering of the levels (e.g., swap strings in a list for a linear progression), remove a level, and add a level. Further, the modification of a level itself is also simple if we assume an adequate level editor.

The overall topic of this dissertation is to address the problems with SLPs while maintaining the benefits. The proposed method, which will be discussed ad nauseam by the end of this dissertation, does not fully satisfy the goal of retaining all the benefits of SLPs. However, the argument that will be made is that the tradeoff of less control, ease of implementation, and ease of modification is both minimal and worth the cost.

1.1 Dynamic Difficulty Adjustment

The proposed alternative to SLPs builds on Dynamic Difficulty Adjustment (DDA) [214]. DDA alters the game based on the player's performance [84]. If the game is too easy for the player, it should be more difficult. If it's too hard for the player, the game should be easier. For example, if the player died thirty times on a boss fight in *Elden Ring* [55],[3] then a DDA system may reduce the boss's health or increase the player's damage [84]; which is to say, make the boss easier to beat. Alternatively, if the player beats a boss on their first try, then the next time they face a boss, that boss should be more challenging.

Another way to view DDA is through a game design lens. Rogers defined a difficult game as one that "promotes pain and loss" and a challenging game as one that "promotes skill and improvement" [143]. A difficult game is not hard to make; a designer can make an easy boss impossibly difficult by making it the size of a pixel, which would make the boss still possible to beat, but nearly impossible to do so. The more interesting problem is to make a challenging game, and successful DDA is about helping designers make their games challenging for as many players as possible.

An example of DDA in a professional game is *Crash Bandicoot* [130].[4]When players died too frequently, they were given a power-up to help them progress, or the spawn point was moved forward. Another adjustment they made was in the *Indiana Jones* [167] style scenario of running

[3]This example may or may not be based on my experience with the game.

[4]https://web.archive.org/web/20250425153504/https://all-things-andy-gavin.com/2011/02/07/making-crash-bandicoot-part-6/

away from a boulder. On failure, meaning the boulder ran over the player, the boulder's speed was slightly reduced to make the scenario easier for the next attempt. Their goal was to "help weaker players without changing the game for the better players."

Developers have different goals for their games. A game like *Elden Ring* is challenging as a feature. The designers wanted the player to fail. A DDA system is not a good addition to a game if the designer does not want the game to become easier. The other direction of a game starting easy and becoming more difficult is also possible, but some designers may not want their game to become more difficult. Consider a walking simulator. DDA could theoretically be added, but should it? Would the designer want DDA? Probably not.

DDA is not a silver bullet [21]. There are many games where DDA should not be used. Even if DDA could improve a game, it may not be the best decision to add DDA to said game. Games are art, and therefore an expression of their creators. Creators decide what is best for their game.

The same is true for the appropriate kind of DDA to add to a game. For some creators, stat adjustments [84] are ideal. Stat adjustments augment a SLP, but the SLP still has to be made. Every level that the players play will always be the same, even if an enemy is different (e.g., in *Crash Bandicoot*, the boulder may become slower, but the player still has to outrun it while dodging obstacles). An alternative to stat adjustment is to have the player play a different level, DDA via level selection. The problem with this kind of method is that instead of designers building one level, they have to build k levels, where k represents different levels of difficulty. This is where Procedural Content Generation (PCG) can help.

1.2 Procedural Content Generation

Before getting into how PCG can be used for DDA, it is important to understand what PCG even is. The goal of PCG is to create content, and that content is created computationally [153]. Meaning, the content was not built by a human designer. The kind of content that is generated and its intended use are generally related to video games, but PCG can be used in other domains. For example, generating an essay falls under the broad umbrella of PCG, where the generated essay is the procedurally generated *content*. Similarly, it is possible to computationally generate music [20], word problems for math [202], videos [197], and more. In the context of video games, levels

[43], attack patterns [101], entire games [33], and more can be procedurally generated.[5]

The word "content" is unideal because it tells the reader nothing about the kind of content that is generated. In the context of this dissertation, the better term is Procedural Level Generation (PLG). The content that a PLG system generates are levels—e.g., a *Mario* level [43, 68, 174, 184, 185].

The way that a PLG system generates a level—or any PCG system generates content—is not prescribed. Generating an essay could be accomplished with a large language model [4, 47], but there are other ways as well [76]. This has led to branches of PCG, such as Procedural Content Generation via Machine Learning (PCGML) [73, 175], which generates content based on examples in a training data set. Another, more recent, branch of PCG is Procedural Content Generation via Reinforcement Learning (PCGRL) [99]. PCGRL generates content through a reinforcement learning process to find the best content [178].[6]

PCGML and PCGRL are both branches of PCG that utilize a fairly complex family of algorithms, but complexity is not required for PCG. For example, let's look at generating the layout of a dungeon. Both methods (PCGML and PCGRL) can accomplish this task, but simpler methods are available. One Constructive PCG [73] method is to randomly break the dungeon space into blocks with a binary space tree.[7] Then, the spaces can be filled in with rooms. The whole space should not be used; otherwise, every room would be directly connected to every other room. Once the rooms have been made, they need to be connected by "carving" tunnels between them. This can be done by using the tree's structure. The result is a PCG algorithm that can generate a dungeon layout randomly.

This falls under a typical style of PCG algorithms used in the games industry, where a "seed" is fed into a random number generator to generate levels. A major benefit of this kind of PCG is that it is fast. Further, given a seed, the same level will be produced every time, allowing for deterministic behavior from a stochastic algorithm. One notable example of this is the world generator in *Minecraft* [127], which allows a user to input a seed to generate the same world every time.

The point of all this, though, is not to say that one branch of PCG is better than another. Each branch is a tool with its pros and cons, and it is up to the developer to decide what is right for their game. For this dissertation, we aren't asking which method is best for a specific game, but

[5] There is much more to the field of PCG than level generation. The following [153, 158, 209, 175] are good starting points for those interested in PCG.

[6] Sutton and Barto provide a strong introduction to reinforcement learning in their textbook [178].

[7] https://web.archive.org/web/20240323005524/https://www.roguebasin.com/index.php/Basic_BSP_Dungeon_generation

which method is best if the goal is to use it for DDA.

PCG for DDA means that we are going to ask the generator to give us levels that match a set of requirements [35, 69, 63]. One approach that would be promising is PCG via search [186] (i.e., search until a level that matches a set of requirements is found by a search method like Monte Carlo Tree Search [174] or a genetic algorithm [152]), but the problem is that finding a level can be slow. In a research setting, it's okay if a system takes two minutes to find a level. However, a game being played by players cannot reasonably ask a player to wait for two minutes between levels.

This problem, though, can be handled if we reduce the size of the problem. Instead of asking a PCG system to generate a whole level, the system can find the right combination of level segments—a level segment is part of a level that can be combined with other level segments to form a whole level—to assemble a larger level [69]. This, though, leads to a new problem: how do we decide what kind of level to assemble?

1.3 Director

A director is a system that attempts to direct the player's experience. One of the most famous examples is the director in *Left 4 Dead 2* [191].[8] It worked by pacing the game into moments of high intensity and relaxation. Crucially, it did not change the game's difficulty; it changed when moments occurred. It could do more, though, like selecting where weapons and enemies spawned, and modify the map to prevent speedrunners from beating a level too quickly.

Another kind of director takes charge of when narrative events occur in a game, also known as a story director [115]. A story director can allow non-player characters to behave more realistically in a game while still moving the player toward the designer's narrative goals. Another approach is a story director via reinforcement learning, where a story director learns the player's preferences and is rewarded for providing that content [211].

In the examples above, the directors have not attempted DDA, but there is no reason they could not. For example, the stats-based DDA approach discussed in Section 1.1 can be seen as a kind of director. The director built for *Left 4 Dead 2* could be augmented to consider difficulty using a similar methodology.

A director, though, can do more than adjust stats, change item spawn locations, and adjust the game's story. A director can direct a PLG system. In doing so, the director takes charge of the level that the player will play, and this can be used for DDA. How, though, can this be accomplished?

[8] https://web.archive.org/web/20120505233022/https://www.valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf

1.4 Research Questions

The first research question, **RQ1**, is about building a director that can direct a PLG system, which works via level assembly, for DDA. The second research question, **RQ2**, deals with how the method built to **RQ1** can be used and whether it is effective with real players.

1.4.1 RQ1: How can usable levels be assembled from level segments for dynamic difficulty adjustment?

The first question you may have had when reading **RQ1** is: What is a *usable* level? The exact definition is provided in Chapter 3, but at a high level, there are two requirements. First, for a level to be usable, there must be a way for a player to beat the level. Second, the level must look and feel like a level in the target game. For example, a PLG system generating levels for *Mario* could generate a level for *Braid* [134], but it is unlikely the level would look and feel like a level in *Braid*.

The next part of the question that warrants discussion is: For what kind of game are these levels assembled for?

The first distinction is between 3D and 2D games. A 3D game is a game where the graphics (i.e., what you see on the screen) are in three dimensions. This allows the player to move along three axes. The first of its kind was *Maze War* [30], released in 1973. More modern games you may be familiar with are *Halo* [23] or *Elden Ring* [55]. A 2D game is a game with graphics in two dimensions. A 2D game only allows the player to move along two axes (horizontal and vertical). It is possible to generate 3D levels, such as the generator for *Minecraft* which uses Perlin noise [137]. However, adding a third axis adds a high degree of complexity to level generation. This has led to ongoing work to improve level generators for 3D games, such as the Generative Design in Minecraft Competition [146, 147]. As a result, the scope is limited to 2D games because the focus of this work is less on PLG and more on DDA.

While there may be exceptions, a 2D game will have one of two camera angles: top-down and side-scrolling. A top-down 2D game has a camera that looks down on the game's environment. This is typically done in dungeon-crawling games, such as *Rogue* [5], where the player can move along both axes freely. A sidescrolling 2D game has a camera that looks at the player from the side—e.g., *Braid* [134], *Mario* [131], *Super Meat Boy* [182], etc. For this research question, both top-down and side-scrolling games will be used.

There are two categories of playtypes for 2D games that are relevant to this work: *turn-based* and *real-time*. A turn-based game is a game where play is separated into turns. This is

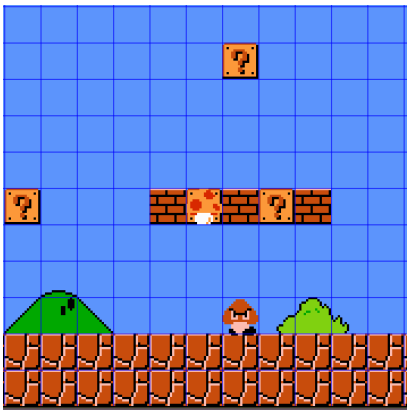


Figure 1.2: Example portion of a level in *Super Mario Bros.* [131] where a grid has been painted on top to show how a level can be separated into a grid.

common in board games and card games where each player takes a turn. In the context of a single-player video game, one common approach is to have a player turn and then an enemy turn, where all the enemies can take an action, such as "move in X direction." A real-time game is a game where everything can move simultaneously. *Mario* is an example of a real-time game where the enemies can move regardless of whether the player is moving. Both types of games are within the scope of the research question.

There are different ways to represent a 2D game level. One way is to have a list of entities with their associated properties, such as a position, which is composed of two floating-point numbers. While a valid approach, generating a level for this kind of format is more complicated than generating a level in a grid-based format. A grid-based game, or *tile-based* game, is one where a grid defines the environment. For example, Figure 1.2 shows a *Mario* level broken into a grid. Each cell in the grid is a specific tile type (ground, solid, enemy, etc.). A grid-based game can have continuous movement, as in *Mario*, or it can have discrete movement where the player moves from one grid cell to another, as in *Rogue*. Ease of generation aside, there is also the second requirement —a level must look and feel like a level in the target game—to consider. The reasons why grid-based is ideal for the second requirement are discussed in Chapter 3. Therefore, only grid-based games (with discrete or continuous movement) are used in answering **RQ1**.

Now that we have specified the kind of game this work is for (a 2D grid-based game that is either turn-based or real-time), the next question is: How is **RQ1** answered? **RQ1** is broken down into two sub-research questions. The idea is that to answer **RQ1**, first **RQ1.1** and **RQ1.2** have to be answered.

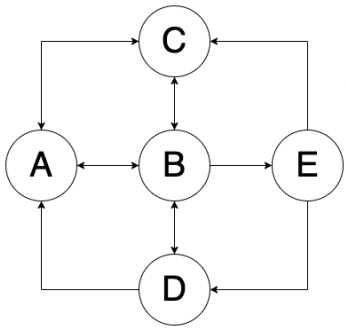


Figure 1.3: Example directed graph. Nodes A, B, C, D, and E represent level segments. The edges represent how these level segments can be combined to form larger levels.

RQ1.1 is: How can usable level segments be generated within an organized structure applicable to dynamic difficulty adjustment? At its core, this question is about PLG. There are many possibilities for how a generator could be built to answer this question that utilize different branches of PCG. The goal, though, is not to provide some kind of taxonomy of all the possible systems with their pros and cons. Instead, a more practical answer is required, and this means that there will be a single method built to answer the research question. That method is called Gram-Elites [16], and is discussed in Chapter 3. One important point to note is that Gram-Elites relies on n-grams—see Section 2.4—to generate level segments

What may be a bit unclear is why a level segment is generated instead of a whole level. Recall from Section 1.2 that level segments can be used to form a level, and that this can simplify PLG for DDA. However, there is a problem that needs to be solved. A level must (1) be completable and (2) look and feel like a level in the target game. Even if the level segments that are put together to make the level meet these requirements, it is not guaranteed that the level will meet them. **RQ1.2** is: How can usable level segments be linked together to form levels? The answer presented in this dissertation is a linking algorithm [17] described in Chapter 4.

The next step is to answer **RQ1** with Gram-Elites and linking, but before we can do that, we need to go over what a *digraph* is: a graph composed of nodes and directed edges. If a node represented a level segment and an edge a link, then that digraph could be used to generate a level by following the edges. For example, in Figure 1.3, say that nodes A, B, C, D, and E contain a level segment. To form a whole level, these level segments can be combined. To see how this is done, let's build a level that is three level segments long, starting from node A. Node A has only one outgoing edge to B. If that outgoing edge also had a link, then that link would be placed between A and B. Moving on to node B, we can see that it has three outgoing edges to C, D, and E. The

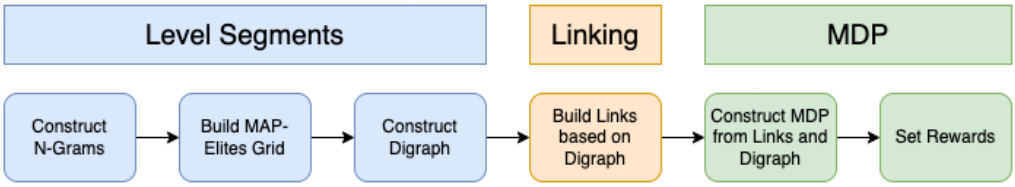


Figure 1.4: This figure shows the system described in this dissertation to generate a Markov Decision Process.

question is: How do we decide which of the three node to select? Random selection is an option, but the better option is to select a level segment based on the player. This is accomplished by framing the digraph as an MDP.[9] This MDP acts as a director, and assembles levels tailored to the player.

With that context, we can now answer **RQ1**. Gram-Elites is used to generate level segments in a structured population, and that structure is used to build a digraph, where each edge is validated with the linking algorithm built to answer **RQ1.2**. That digraph is converted into a MDP, and the MDP generates levels. The whole process built to answer **RQ1** can be seen in Figure 1.4. What is missing, though, is how the system works with players, and that takes us to the second research question.

1.4.2 RQ2: Does an MDP built automatically, a handcrafted MDP with designer-set rewards, or a static progression result in a better player experience?

This research question evaluates the system built to answer **RQ1**. Before **RQ2** can be answered, though, there are, again, two research sub-questions. Both questions—**RQ2.1** and **RQ2.2**—are about finding the right reward for the MDP. The reward guides the MDP and therefore can be thought of as the guide to the director. The wrong reward would make the system ineffective.

The first research sub-question is **RQ2.1**: How well do computational metrics approximate difficulty and enjoyment? In this work, a computational metric is an algorithm that can assess a level. "Assess" is purposely vague, but one restriction is that the output of the metric must be numerical. One simple example is to return the number of enemies in a level. A slightly more complex computational metric is *density*, defined in this work as the number of solid blocks in a level divided by the total number of grid cells in a grid-based level. There are many more computational metrics, and it was/is infeasible to evaluate every one of them. Therefore, only a subset of computational metrics was used to try to approximate difficulty and enjoyment.

[9] MDPs are discussed in detail in Section 2.6.

This leads to the question, why do we want to approximate difficulty and enjoyment? The hypothesis was that using one or the other as a reward for the MDP would result in a good player experience—see Chapter 8 for more on what a "good player experience" is—but first we had to gather player ratings of both difficulty and enjoyment of level segments with a player study to do so. Those ratings, though, can be used for more than just a reward for the MDP—and they will be used for the next research question—they can also be used to find which computational metrics best approximate difficulty and enjoyment.

This leaves one last question: Why do we want to approximate difficulty or enjoyment with computational metrics if we have player ratings? The question itself is interesting (in my opinion), but that doesn't motivate why it should be part of this dissertation. The reason why it is a necessary part of this dissertation is as part of a "what-if." In this case, what if, for example, having the reward of the MDP be based on the difficulty of a level segment resulted in the best player experience? The requirement then for any future users of this work would be that they run a player study to gather ratings of their level segments. The likelihood that any developer would do this is low, and, as a result, it would be unlikely that they would try the system themselves. The point, therefore, is to provide an alternative to player ratings. Additionally, it is always possible for designers to use a computational metric to initialize the rewards table for the MDP, and then modify the rewards by hand after—see Chapter 9 for more on this topic.

The second sub-research question, **RQ2.2**, is: Is it better to optimize for difficulty, enjoyment, or both, in terms of player experience and dynamic difficulty adjustment, when assembling levels? Unlike the work done to answer **RQ1**, this research question required a study with real players, and, as a result, the player study was the first test of an MDP assembling levels for DDA with real players.

The question itself describes two of the conditions in the player study, with, for example, optimizing for difficulty meaning that the reward table of the MDP was built from the difficulty ratings gathered to answer **RQ2.1**. The term "both" is less clear. It could mean the mean of difficulty and enjoyment, or it could mean something else, like switching between the reward tables for difficulty and enjoyment every three levels played. In this case, "both" refers to both examples, resulting in two conditions. The last, and final condition in the study used the same MDP, but levels were assembled randomly, and this condition was a baseline.

Again, a "good player experience" is is defined in later chapters, but what does better DDA mean? This chapter contains many variants of a phrase that describes the answer: "We want a level that is neither too easy nor too hard." That, though, is a feeling, not a definition. A definition,

for example, is that a successful DDA via PLG system is one that resulted in a 60% level completion likelihood for the target player [62], and, therefore, the best condition in terms of DDA would be the one that resulted in the closest completion likelihood to 60%. The definition, though, is stringent, and I doubt that 60% is ideal for every game. Therefore, a more holistic view of DDA was taken, and all the data was analyzed to examine what was too difficult for players and what was not. After analysis, we found that the condition that used the mean of the player ratings of difficulty and enjoyment performed best.

Now with the two sub-research questions out of the way, we can return to **RQ2**: Does an MDP built automatically, a handcrafted MDP with designer-set rewards, or a static progression result in a better player experience? To answer this research question, two player studies were run. One for a roguelike (which was used when answering **RQ2.2**) and one for a platformer. (Section 2.2 provides definitions for what a roguelike and what a platformer is.)

The original intention was to use the player ratings gathered to answer **RQ2.1** as the reward table of the MDP for the first condition ("an MDP built by the system") of the roguelike player study. However, as will be explained in Chapter 9, a new MDP was made instead. The rewards for this MDP were built from a linear regression model that estimated the player ratings to estimate the mean score. The rewards for the MDP used by the platformer were also estimated, but the model could not be used since it was built to evaluate levels for a specific roguelike.

The second condition of both player studies used the same MDP from their first condition, but the reward table was not based on player ratings. Instead, the reward table was based on the node's depth, or the distance from it to the start node. This condition was to test whether specific rewards were necessary or if the simplest reward, which got the player to the end of the game as fast as possible, while still being focused on DDA, was best.

The third condition was an MDP built by hand. This was done to (1) display how it could be done, (2) show designers who did not want to use the system shown in Figure 1.4 that they could still get DDA via PLG without it, and (3) act as a comparison to the automatically generated MDP.

The final condition used the automatically generated MDP and ran a breadth-first search from the first node in the digraph to the last node. The path was stored, and then this acted as a SLP.

These conditions were used in both player studies to determine whether the overall approach worked. This breaks down into multiple subquestions, like whether DDA worked or did not, and whether the approach was better than a static progression. Overall, the results were clear that MDP-based level assembly worked. The SLP for the roguelike was too easy to play, and the SLP for the platformer was too difficult. The best performing condition, overall, was not determined

based on the player experience because the measured player experience showed little to no differences between all three MDP-based conditions. Based on the overall win rate across the conditions, though, it was clear that the MDP built by hand was the best performer for the designer's desired player experience.

1.4.3 The Questions

- **RQ1**: How can usable levels be assembled from level segments for dynamic difficulty adjustment?
  - **RQ1.1**: How can usable level segments be generated within an organized structure applicable to dynamic difficulty adjustment?
  - **RQ1.2**: How can usable level segments be linked together to form levels?
- **RQ2**: Does an MDP built automatically, a handcrafted MDP with designer-set rewards, or a static progression result in a better player experience?
  - **RQ2.1**: How well do computational metrics approximate difficulty and enjoyment?
  - **RQ2.2**: Is it better to optimize for difficulty, enjoyment, or both, in terms of player experience and dynamic difficulty adjustment, when assembling levels?

1.5 Dissertation Statement

A Markov Decision Process—which can be created automatically or by hand—can act as a director for DDA via PLG for 2D grid-based games, and the final result is a positive player experience.

1.6 Outline

- Introduction
  - Chapter 1 introduces the dissertation and is the chapter you have almost finished reading.
  - Chapter 2 discusses material relevant to many future chapters. It begins by introducing game genres and some key distinctions relevant to one particular genre: roguelikes. After, expressive range is introduced [162, 172]. Many PCG researchers use expressive range to evaluate content generators. N-grams [91] are presented next. N-grams

significantly influence all the subsequent work. The following section discusses what a Genetic Algorithm (GA) is, including genetic operators, and then goes into MAP-Elites [129], which generates level segments in this work. The final part of this chapter covers MDPs and the math that makes them work.

- **RQ1**
  - Chapter 3 addresses **RQ1.1** and discusses the first part of the system in Figure 1.4: level generation via PLG. It shows one way that level segments can be generated with MAP-Elites (see Section 2.5.1). It introduces novel mutation and crossover genetic operators (see Section 2.5 for information on genetic operators) that uses n-grams. The method presented showed improvements over standard genetic operators when generating level segments for three games.
  - Chapter 4 addresses **RQ1.2** and shows how level segments can be combined to form larger levels. Concatenation has been a common approach, but there are no guarantees that the resulting level will be playable [154, 159]. Further, if one level segment starts or ends with a structure (e.g., a pipe in *Mario* [131], also see Chapter 4 for more info on structures), concatenation does not ensure that the resulting level will have no unbroken structures. The alternative presented is *linking*. Links, built using n-grams and an agent, serve as bridges between level segments. These links guarantee that the final level is complete and has no broken structures.
  - Chapter 5 addresses **RQ1** and brings in the work from the past two chapters to form an MDP. This MDP acts as a director and decides which combination of level segments the player will play next. The evaluation compares three different directors. The rewards are heuristics that approximate difficulty. Agents then evaluate the resulting director. Some agents are very good at the game, others are bad—some like challenging levels, and some like easy ones. The results show that the approach built to answer **RQ1** works in theory. The best director even works when agents of different types are swapped during a gameplay session.
  - Chapter 6 introduces a tool called *Ponos*, which combines the work done to answer **RQ1** into one tool that developers can use. It shows what developers would need to do to use *Ponos* and discusses what they would need to do to use an MDP director in their game.
- **RQ2**

  - Chapter 7 addresses **RQ2.1** and introduces an online player study run for a roguelike game called *DungeonGrams*. Players were asked to assess levels based on difficulty and enjoyment. The results were then explored using eleven heuristics to test whether difficulty and enjoyment could be estimated well. Difficulty was reasonable to approximate, but none of the eleven heuristics tested could correctly estimate enjoyment. Further, there was little correlation between difficulty and enjoyment. In addition, two heuristics that utilized Jaccard Similarity [86] were introduced.
  - Chapter 8 addresses **RQ2.2**. A player study was run with *DungeonGrams* that used the level ratings gathered in the previous chapter. The study had five conditions with 250 participants total. We found that the *mean* condition, which was the mean of the difficulty and enjoyment player rankings, resulted in the best DDA. There were almost no notable differences in the players' overall experience, though, for the MDP-based conditions in the study.
  - Chapter 9 answers **RQ2**. Two player studies were run, one with *DungeonGrams* and another with a platformer called *Recformer*. Both player studies ran with a total of 160 participants and four conditions. Two of the conditions were built automatically with *Ponos*, one condition used the generated digraph from *Ponos* to construct an SLP, and the final condition used a handcrafted MDP. In terms of subjective level competition rates, the handcrafted condition performed best, but there were statistically significant differences between the player experiences of the conditions.

- Conclusion
  - Chapter 10 discusses what was learned from using *Ponos* to build two MDPs for Chapter 9. It is not a guide on how to use *Ponos*, though. Instead, it is meant to give developers a better insight on the experience of using *Ponos*.
  - Chapter 11 is the final chapter of this dissertation, and it summarizes the results and provides some closing remarks with interesting areas of future work.

Chapter 2

Background

> He was waiting to do what he knew he had to do, and he was waiting with a dread and an anger and a sorrow that grew more intense with every minute that passed.
>
> John Williams, *Stoner*

This chapter introduces the algorithms and concepts that later chapters use. While I have spent more time on these sections than I care to admit, I recommend reading the section titles and only reviewing the material if you are unfamiliar with the topic.

2.1 Agent Evaluation

This work deals with game content for players. Therefore, it should be self-apparent that the best evaluation method involves players (i.e., real people). However, player evaluation is not always prudent. Player evaluation requires a game that is ready for players. Making such a game can be extremely time-consuming. The developer has to program the game, make the art assets, create sound effects, and put everything together into one cohesive experience that is polished enough to be worth a stranger's time.

One example consideration is that the game's graphical fidelity can affect the player's experience [120]. The kind of game evaluated affects the required graphical fidelity. For example, some casual games need not have the best graphics for proper evaluation [60]. These kinds of considerations make making a game that players will play and evaluate difficult and time-consuming.

Given a game, the next step in player evaluation is to run a player study. This typically involves multiple conditions (i.e., multiple versions of the game). If the study is online, the game should run in a browser. This also means that the researcher has to set up a system for getting logs from the player's play session. This typically means that a server is required. In addition, regardless of whether the study is online or not, there should be, ethically speaking, compensation for the labor of participants.

These considerations and many more are why player evaluation may not be prudent. The alternative is *computer evaluation*. With computer evaluation, a researcher evaluates content based on a set of heuristics or computational metrics [14, 193, 117, 195, 198]. One example is expressive range analysis [162, 172] (see Section 2.3).

A subarea of computer evaluation is agent evaluation. In agent evaluation, an agent, an Artificial Intelligence (AI) system, plays a game and generates a playthrough. The evaluation uses the playthrough. The evaluation can be simple, such as testing whether a level can be completed, or it can be more involved, where heuristics use the agent's playthrough as input [14, 70]. However, it is important to understand that agent evaluation is not a silver bullet. Making a game-playing agent can be a time-consuming and difficult process [125, 184].[1]

Importantly, computer evaluation avoids many of the problems of player evaluation listed above. As a result, many of the difficulties involved in making a game for a study are no longer relevant. An agent does not need graphics, just the game's state. An agent does not need payment, just electricity. An agent can evaluate content almost instantaneously. They can play thousands and thousands of levels. Difficulties like setting up a server for logging are irrelevant. Sound design is unnecessary.

The work presented in this dissertation utilizes both computer and player evaluation (including but not limited to agent evaluation). The reasons for each choice are discussed when relevant in the subsequent chapters.

2.2 Game Genres

This work uses several games as testbeds for evaluating the proposed techniques. While there are many genres, the decision was made to focus on two: platformers and roguelikes.

[1] Millington's textbook is a great starting point for those interested in AI for Games [125].

Figure 2.1: Taken from Braid's press kit, this screenshot shows an example lock and key puzzle. The player has to grab the key first (right side of the level) and then take it back to the locked door (middle-left) which is blocking the puzzle piece. If the player tries to go through the door without the key, they will not pass.

2.2.1 Platformer

A platformer is a 2D side-scrolling, real-time game where the player runs and jumps in continuous space. The most famous platformer is arguably *Super Mario Bros.* (aka *Mario* [131]), which features a plumber trying to save a princess. The plumber, Mario, has to travel from left to right in levels and jump between platforms. The goal is to make it to the end of each level without dying. While there are additional elements like coins, enemies, power-ups, bosses, etc., the core is running and jumping, or platforming which is where the term platformer comes from.

A platformer does not have to be only horizontal; a platformer can also require the player to travel down (e.g., *Downwell* [56]) or to travel up (e.g., *Kid Icarus* [132]). Further, a platformer does not have to have the goal of running from one end of the level to the other side. A common design in games is called "Lock and Key," and that can be part of the platformer, where the player has to collect keys before they can beat the level (e.g., *Braid* [134] uses this to make it more difficult for players to grab puzzle pieces which are required to progress through the game, see Figure 2.1).

2.2.2 Roguelike and Roguelite

A roguelike is a 2D top-down, turn-based game where the player moves from grid cell to grid cell. The term "roguelike" comes from one of the first games of the genre, *Rogue* [5]. For this

work, we will use common features to define the roguelike genre: procedural content, permanent death, turn-based, single-character with an inventory, and discoverability [85]. However, it should be noted that the whole community does not agree with this definition. Games like *Hades* [177] and *The Binding of Isaac* [121] feature a *meta-progression* where actions in one gameplay session can have consequences in later sessions. In *Hades*, for example, the player can collect "darkness" during a run of the game. When the player dies in a run, the collected "darkness" does not go away, and it can be spent to upgrade the player's character. This leads to the argument that a meta-progression violates the concept of permanent death, leading some to term games that violate the stringent definition of a roguelike as a *roguelite*.

The work presented in this thesis does feature a kind of meta-progression in that the results of past lives affect the levels shown to the player in later lives to achieve dynamic difficulty. As a result, any game that uses the approach presented in future chapters cannot be a roguelike and is, therefore, a roguelite.

2.3 Expressive Range

Imagine you asked a PCG system to generate one thousand bowls of oatmeal; no fruits, chocolate, cinnamon, or anything else, just a bowl of oatmeal. The system then outputs all the bowls. At a glance, it is unlikely that you'll be able to see any difference between bowls. This hypothetical system will generate computationally distinct bowls, such as one oat rotated 180 degrees or slightly less cooked than before. Still, there will be little to no difference to the eye. This is known as the "1000 bowls of oatmeal" problem, and it is a constant challenge for any PCG system [31].

When building PLG systems, it has been commonplace in academia to assess the kinds of levels that the system can generate. The most common approach is to analyze the *expressive range* of the generator. Expressive range attempts to "capture the style and variety of levels that can be generated" [162]. This is accomplished with computational metrics—also known as a Behavioral Characteristic (BC). One example of a computation metric is *density*. Density is the number of solid tiles divided by the total number of tiles in a grid-based level. Therefore, if one were to analyze the expressive range of a PCG system based solely on density, the PCG system would generate n levels. Those levels are then analyzed in terms of density and plotted onto a line between 0 and 1. Commonly, a grid-based approach is adopted where the programmer could assign k buckets. If $k = 2$, then there would be two buckets for different ranges of density ([0.0-0.5), [0.5-1.0]). Levels are then placed into buckets based on their density. Of course, in practice, you would want a finer

resolution. Regardless of the k, the final plot will have each bucket be darker or lighter based on how many levels were generated that fit their bucket. This provides a good way to quickly see the expressive range of a generator. For example, the plot may show that the PCG system cannot generate levels of low density.

The problem with the example presented is that one dimension is often insufficient. Smith and Whitehead's paper presents a two-dimensional hex-based graph with two computation metrics instead [162]. This provides a better view of how expressive a PCG system is. However, there are still problems with this. The first is that the computational metrics chosen may not be a good indicator of variety to the player. Going back to the oatmeal example, the metric may capture the number of grains changed, but how noticeable will that "variety" be to the player? Computational metrics must be chosen carefully for expressive range to be useful. A second problem is that the graphing method is limited to two dimensions (two computational metrics). The solution of graphing multiple two-dimensional plots, while valid, is less satisfactory as we lose the bigger picture in the process.

In the work presented in later chapters, expressive range as presented is used. The lack of strong visualization techniques is why the work in Chapter 3 is limited to two dimensions. However, I have become aware of other work in the area. Specifically, Summerville has a paper on the topic [172] which not only presents extensions to improve expressive range but also raises the point that expressive range is not the end-all be-all of methods to analyze PCG systems. The field of analyzing PCG systems is underexplored, and expressive range should be seen as the beginning of work to be done and not the end. Still, it must be said that one of the strengths of expressive range is its simplicity and ease of implementation in research projects.[2]

2.4 N-Gram

The original motivation behind the n-gram was to calculate the probability of one word following a set of words (i.e. a prior) [91]. For example, what is the probability that "Bond" occurs after "James"? Using "James" as a prior is an example of an n-gram where $n = 2$. If $n = 3$, then there would be exactly one word before "James."[3] To find the probability, n-grams read a corpus of text and count occurrences with a sliding window based on n.

[2]For those interested in the field of evaluating PCG systems, a more recent paper from Withington et al. presented a survey on the topic for PLG [204].

[3]A one-gram is also called a uni-gram. With $n = 2$, you get bi-gram. $n = 3$ is a tri-gram.

To see the n-gram in action, let's look at a simple sentence: "The dog and the person went out for a walk." If we use $n = 1$, then the calculation does not rely on a prior. For example, $P(\text{"and"}) = 1/10$. This is found by counting how many times "and" occurs in the sentences ($N(\text{"and"})$) and dividing the result by the number of words ($N() = 10$). The calculation for "the" is different because it occurs twice, resulting in $P(\text{"the"}) = 2/10$.

A uni-gram does not have the context of prior words, but any n-gram where $n > 1$ will use prior words to calculate the probability of the next word. For example, a bi-gram can calculate the probability of the word "person" when it follows the word "the": $P(\text{"person"}|\text{"the"}) = 1/2$.

Instead of calculating the probability of a word following another word, we can also use an n-gram to generate a sentence. For example, we'll use a unigram trained on the example sentence above to generate a sentence starting with the word "went". A uni-gram doesn't rely on any prior, so any of the nine words from the example sentence are fair game to be generated after "went". The most likely word to be generated is "the" with a 2/10 chance, but that wouldn't be great output: "Went the." But more words could be generated, and you may end up with something like: "Went the dog person went the a walk."

This is where an n-gram with $n > 1$ can be useful because they have context through the prior. If we started generating from "went" with a bi-gram, the bi-gram generates "went out for a walk." That is because there is no other associated output with those four words as a prior. However, notice the potential problem with the word "walk": There is no output for "walk": $N(\_|\text{"walk"}) = 0$. This is because there is no word in our example sentence that follows the word "walk." This is known as an *unseen prior*. Besides adding additional training data, there are several solutions like smoothing [28] and the backoff n-gram [97], but they are not used in this work so I won't go into detail on them. An alternative is fully connecting the n-gram, and this is discussed below in Section 2.4.1.

A key question when using n-grams to generate anything is: What is the right size of n? From the example above, it is hopefully clear that a uni-gram gives weighted random behavior while a large enough n effectively memorizes the input. The answer also depends on the size of the input. In the example sentence from above, we saw that a tri-gram can only output what was input, but if the input was the entire corpus of Shakespeare's work, then a tri-gram would be far too small to memorize and faithfully output any full work.

N-grams can be used for more than generating sentences, though. Dahlskog et al. generated *Mario* levels with an n-gram and analyzed several n values [43]. As input to the n-gram, they broke a Mario level into vertical slices—see Figure 2.2—but note that multiple levels can be used as

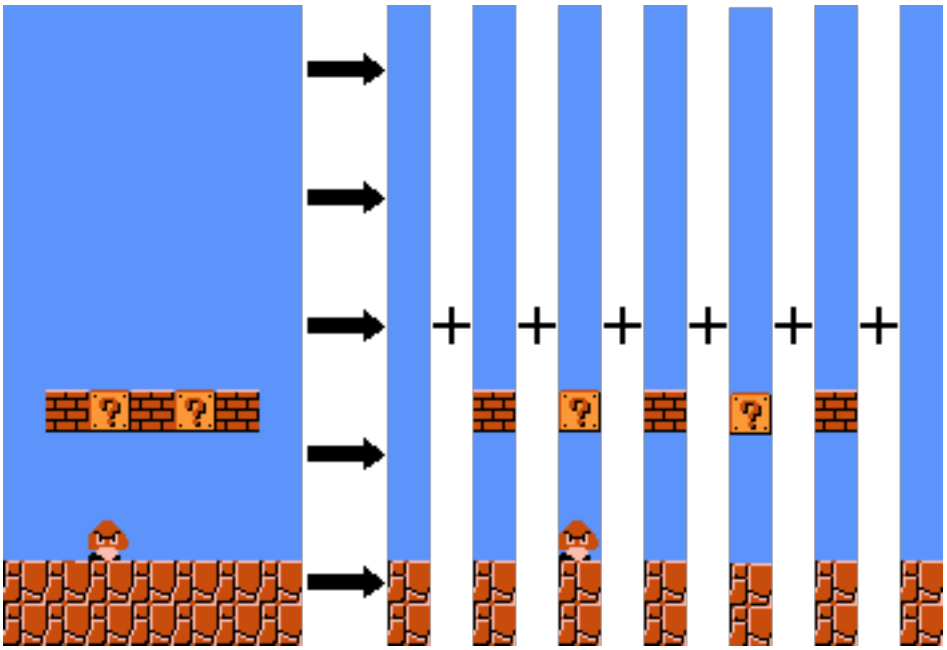


Figure 2.2: This is a demonstration of how a *Mario* level can be broken up into a set of vertical slices. These level slices can then be used to generate new *Mario* levels.

input. Dahlskog et al. found that "the effects of varying n are rather drastic. Essentially, unigrams produce a haphazard mess, bigrams produce some local structure with much repetition and trigrams produce levels with good local structure that are stylistically similar to the training corpus."

While a common use of n-grams for games is procedural content generation [43, 163, 174, 164, 16, 17], they have also been used for game playing agents [179, 125], player modeling [105], and more [194].[4]

2.4.1 Strongly Connecting an N-Gram

A *graph* is made of nodes and edges. A *node* is the building block of a graph and, in the context of computing, has some data associated with it. In the context of n-grams, every prior is a node. An *edge* represents a connection between two vertices. If the graph is a *directed graph* (i.e., digraph), an edge will have a given direction with a source and a target node, whereas a graph assumes bi-directionality. An edge can also be associated with data. In the context of an n-gram,

[4] A fun example is to use a backoff n-gram as a gameplaying agent for *Rock-Paper-Scissors*: `https://github.com/bi3mer/rock_paper_scissors`. The n-gram predicts what the player will play (rock, paper, or scissors) based on what the player has done in the past, and then a winning action is selected based on the prediction (i.e., the n-gram predicts rock, so paper is played).

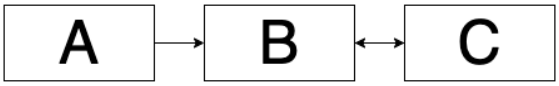


Figure 2.3: Example of a directed graph that is not strongly connected. Node A cannot be reached by nodes B and C. If node A were removed, the graph would be strongly connected.

an edge can have the output word and the probability of that word given the prior. Edges are how we can calculate the *degree* of a node, which is the number of edges associated with a given node. A node with a degree of one is referred to as a *leaf*. In the case of a directed graph, a degree can be further specified to *outdegree* and *indegree*. Outdegree is the number of outgoing edges of a node. Indegree is the number of incoming edges to a node.

A *strongly connected graph* is a graph where every node is reachable from every other node. As discussed above, an n-gram can be viewed as a directed graph where priors are vertices and edges are tokens that can occur after a given prior. If an n-gram is not strongly connected, there is guaranteed to be at least one unreachable prior from a specific start prior.

This can be a problem in the context of procedural content generation. Let's say that we have an n-gram that we know is not strongly connected, and the designer wants a level that is n vertical slices long. The n-gram may generate n level slices, but because it is not strongly connected, it may reach an unseen prior and fail to generate a long enough level. The simplest solution is to run again with either the same start prior—relying on randomness for a different path to be taken, which assumes that the outdegree of at least one prior along the path to have an outdegree greater than one—or generation could start from a different start node. This is known as a *generate-and-test* approach [185]. This approach assumes wasted computation as part of finding an output that matches the required criteria. An alternative is to run a search through the n-gram, such as Monte Carlo tree search (MCTS) [174]. However, the work discussed in Chapter 3 requires a strongly connected n-gram.

Testing whether a bidirectional graph is strongly connected is simple. A tree search (depth-first or breadth-first) runs from any node, and the graph is strongly connected if every node is found. Otherwise, it is not strongly connected.

This approach does not work if the graph is directed. A simple example to demonstrate this is the graph seen in Figure 2.3. If a tree search were to start from node A, the search would return that the graph was fully connected. However, a search starting from B or C would show that the graph is not fully connected. This leads to a rather slow solution: running a tree search from

every node and stopping if any search finds a node that is unreachable. Of course, this could be sped up by making every tree search run in parallel, but there are far better solutions. The solution used in this work is called Tarjan's Strongly Connected Components Algorithm [181]. The details of the implementation are out of scope for this thesis.[5] It is important to understand that Tarjan's algorithm returns the largest strongly connected subgraph, where the subgraph can be the original graph if the original graph is strongly connected.

2.5 Genetic Algorithms

The idea behind how a GA works was inspired by Darwin's theory of natural selection [45]: the strong survive and the weak do not. In the context of optimization, solutions that perform well survive, and solutions that do poorly do not.

The overview of the algorithm is simple. First, generate a population of solutions randomly. In the context of this work, a level is represented by an array of level slices. Random generation of a level, then, is a random selection of level slices that have been placed into an array of some pre-determined size.[6] For convenience, we will also calculate the fitness of each solution after generation. Fitness can be something simple like the percentage that an agent can make it through a level; 100% means the agent beat the level. The goal for the GA, given this fitness function, is to generate a completable level.

At this point, a GA will loop to optimize the population (how this is done will be covered in a moment) until a stopping criterion is met. The criterion could be a set number of epochs (i.e., run for n loops). For this example, another stopping criterion is when a level with fitness of 100% is found. If, though, there was reason to believe that it was unlikely that a level that can be completed was not generable given the set of level slices, then using the stopping criterion of finding a perfect level is a bad idea because the GA may never finish running. Instead, a combination of the two should be used (i.e., run until a completable level is found or the maximum number of epochs have occurred.)

Inside the loop, a new population is generated by a set of *genetic operators* which produce new solutions from the previous population. The first operator is the *selection operator*. This operator selects members of the old population for the new population. This can be done randomly,

[5] I have a Python implementation on GitHub (https://gist.github.com/bi3mer/0e107186911a5457d6eb32805b513a05) for those that are curious.

[6] Genetic algorithms do not require that the array or level be the same size, but the work presented in this thesis assumes the same size.

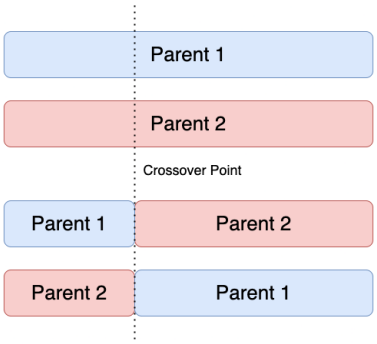


Figure 2.4: Example of single-point crossover. Elements are swapped between the two parents based on a crossover point. Importantly, opposite sides are swapped, else one side of both solutions would always be the same.

but it is better to use a weighted random selection such that better-performing solutions are more likely to survive between generations.[7] Typical implementations of the selection operator will select solutions two at a time to accommodate the second operator, the *crossover operator*. There are many different kinds of crossover operators [189], and to cover them all is out of scope. The first step is to generate a random number between 0 and 1 and test if that number is less than the programmer's assigned crossover likelihood. If not, the crossover operator will return the two input solutions unmodified. Otherwise, crossover modifies the two input segments. The simplest crossover operator is *one-point crossover*. This works by randomly selecting a point for which the elements to the left in the first solution are swapped for elements to the right in the second solution, see Figure 2.4. The third and final operator is *mutation* [136]. This operator will randomly modify individual elements of a solution. Again, the programmer has to configure a likelihood for mutation to occur. You will typically see high crossover likelihood and low mutation likelihood, but mutation is essential because it introduces new elements into the population. Some work has played with this idea by modifying the mutation probability at run time with simulated annealing [205].

These three operators run in a loop until the stopping criterion is met. It is common to return the best solution at this point, but this is a mistake in some contexts, such as level generation. If using epochs as the stopping criterion, there is likely a whole set of levels in the population that are completable and, therefore, usable. That being said, we have done nothing in this example to guarantee that the levels are different from each other. In other words, the *expressive range* [162] of an evolutionary-based generator is unclear.

[7] Another version is a pre-selection of the best n solutions, which are guaranteed to be in the next population without any modifications.

2.5.1 MAP-Elites

A problem with the method we just explored is that the search isn't guaranteed to produce a final population that exhibits different BCs (for example, all the levels generated may be very flat). This isn't necessarily surprising because the GA, as defined, wasn't tasked with generating a diverse population; it was tasked with finding the best solution. Quality-Diveristy (QD) algorithms [140] make the argument that evolutionary computation is best for diversification, not optimization. Through that diversification, QD algorithms can also achieve quality.

Multi-dimensional Archive of Phenotypic Elites, MAP-Elites for short, is a QD algorithm [129]. It uses the same genetic operator setup as described above, but there are differences in the structure of the population and the selection operator.

Starting with the population, MAP-Elites does not use an array to store solutions and their respective fitness.[8] Instead, the population is stored in an n-dimensional grid. Each dimension represents a BC. The grid is tessellated such that each dimension is split k_i times (e.g., if the min of the BC is 0, the max is 1, and $k = 4$, then the split ranges are $0 - 0.25$, $0.25 - 0.5$, $0.5 - 0.75$, and $0.75 - 1.0$), where i represents the fact that each dimension can be split differently. Given two BCs, this results in a rectangular grid-structured population; given three, a cuboid; etc. Inside each *bin* or *cell* are populations. However, these populations are size-limited. This is where the *elite* in MAP-Elites comes from. When a solution is placed into a cell—the cell which is based on the solution's BCs—the placement breaks down into three cases:

- If there are fewer than n elites in the cell that corresponds to the solution's BCs, then the solution is placed into the corresponding grid cell.
- If the fitness of the worst performing elite in the cell is less than or equal to the fitness of the new solution, the old elite is dropped, and the new solution is placed into the corresponding grid cell.[9]
- Otherwise, the newly generated solution is dropped.

The second difference is the selection operator. Before, we had an easy way to run a weighted-random selection since the population structure was perfect. The structure of the new population makes this more difficult with multiple arrays of arrays with inner populations that are also arrays. Many selection operators can be used for MAP-Elites [49], though. In our case, we'll

[8] Some implementations may have a separate array for fitness.
[9] Ties are in favor of new level segments.

discuss the simplest approach: random cell selection. Once two bins have been selected, it is as simple as running a weighted random selection from the elites. Then, crossover and mutation operators can run, and the new solutions can be placed into the grid.

This, though, shows another difference between MAP-Elites and genetic algorithms. Genetic algorithms will typically run for a set number of epochs, replacing the whole population each epoch. MAP-Elites modifies its population in place with each iteration. The problem is, what is the stopping criterion for MAP-Elites? The solution is to either have a custom stopping criterion (e.g., run until a completable level is generated) or to run for a set number of iterations.

2.6 Markov Decision Process

This section goes into the math behind Markov Decision Processes (MDP). If you find code easier to follow than descriptions of math, I recommend looking at either my Python[10] or TypeScript[11] implementation. Both implementations are used in different parts of this work. Note that the implementations are graph-based rather than matrix-based.[12]

An MDP is a framework for modeling decision-making for discrete-time and state environments [178]. An MDP is made up of:

- A set of states S.
- A set of all possible actions A, and a subset of actions available at any state s, where $\forall s \in S : A(s) \in A$.
- A set of rewards $R \in \mathbb{R}$, and the rewards $R(s)$ at any state s, where $\forall s \in S : R(s) \in R$.
- A transition model $\forall s \in S, a \in A(s) : P(s'|s, a) \in P$, where $P \in \mathbb{R}$ is in the range of $[0, 1]$ and is normalized.[13]

Different definitions may or may not include a policy in the list above. A policy $\pi(s)$ outputs an action a. A random policy is built by setting a random action for every state s, see Equation 2.1. Note that the "random" chooses an action from the set of actions available at state s as defined by the action table, $A(s)$.

[10] `https://github.com/bi3mer/GDM`

[11] `https://github.com/bi3mer/GDM-TS`

[12] Matrix-based implementations of MDPs are easier to implement than graph-based, but, more importantly, a matrix-based MDP will also be likely more efficient because developers can rely on libraries with highly-optimized code [201, 13], such as NumPy [77]. I discuss why a graph-based implementation was used in Chapter 5.

[13] For clarity, the transition table probabilities for every state s should add to 1.

$$\forall s \in S : \pi(s) \leftarrow \text{random}(A(s)) \tag{2.1}$$

The goal, though, is to find an optimal policy π^*. "An optimal policy is a policy that yields the highest expected utility" [145]. We can represent utility as another table: $\forall s \in S : U(s) \in U$, where $U \in \mathbb{R}$.

$$\pi^*(s) = \arg\max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s') \tag{2.2}$$

Given the correct utility values, the optimal policy can be calculated with Equation 2.2. However, we have to calculate those utility values.

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s') \tag{2.3}$$

The utility of a state can be found with the Bellman equation, see Equation 2.3. γ is a discount between 0 and 1. When γ is close to 0, the calculation results in decision making where future rewards are unimportant. When γ is close to 1, future rewards are important. Therefore, γ reflects the environment and what the agent values.[14]

A problem with the Bellman equation is that it is recursive, where the utility of one state depends on its connecting states, and the connecting states of the connecting states, and so on. The problem becomes intractable at scale. This can be solved with value iteration or policy iteration.

2.6.1 Value Iteration

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U_i(s') \tag{2.4}$$

Value iteration uses the Bellman update, see Equation 2.4. The difference between the Bellman equation and the update is that the Belmman update creates a new utility table (U_{i+1}) via iterations without a recursive call; it uses the previously calculated value of $U_i(s')$—typically, the utility table is initialized with zeroes—and relies on iterative updates to converge to the correct utility values for each state. The proof of convergence is out of scope, but a simple version can be found in the textbook *Artificial Intelligence: A Modern Approach* [145].

[14] In the context of using an MDP for level generation, see Chapter 5, an interesting idea is to play with γ based the player. If an algorithm can figure out the kind of player, short-term versus longer-term thinking, γ could be adjusted to reflect that player.

To stop value iteration, there is a variable ϵ that needs to be set by the programmer. During iteration, the difference between the utility of each state and the new utility is calculated: $|U_{i+1}(s) - U_i(s_s)|$. If this value for all states is ever lower than $\epsilon(1-\gamma)/\gamma$, value iteration is complete and the optimal policy can be calculated with a modified Equation 2.2 that uses U_{i+1}. An alternative approach is to simply set a maximum number of iterations and terminate regardless of convergence. After all, unless one is interested in the exact utility of a state, all that matters is finding an optimal policy.

2.6.2 Policy Iteration

$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s'|s, \pi_i(s)) U_i(s') \tag{2.5}$$

Policy iteration loops over two steps: (1) policy evaluation and (2) policy improvement. Policy evaluation is accomplished with Equation 2.5. The difference from value iteration's Bellman update is that policy evaluation does not use the max operator for all actions and instead only calculates for one action: $\pi_i(s)$.[15] Policy improvement calculates a new policy π_{i+1}, see Equation 2.2 but use U_{i+1}. During this calculation, a diff is performed to compare π_i and π_{i+1}. If there are no changes from the previous iteration to the current iteration, then $\pi_{i+1} = \pi^*$, and policy iteration is complete.

Policy iteration does not guarantee that the utility values calculated are the same as what value iteration finds. However, the policies will be the same, given a sufficiently low ϵ for value iteration. The proof of convergence of policy iteration is again out of scope, but the aforementioned textbook [145] is a good starting place.

2.6.3 The Decision to Use Policy Iteration

In Chapter 5, the idea of using an MDP to assemble levels is introduced and explored. Importantly, value iteration was not used, whereas policy iteration was. This decision was based on (1) simplicity and (2) speed.

Policy iteration is simpler to use than value iteration. The main difference is that value iteration requires the user to define ϵ. If ϵ is too large, there is no guarantee that the final policy will be the optimal policy. If ϵ is too small, the algorithm can take an extremely long time to run

[15] π_0 is assigned with random actions as shown in Equation 2.1.

and waste compute cycles when an optimal policy has already been found. Therefore, in practice, ϵ must be determined experimentally when used in real-time applications. This consideration is, at best, an inconvenience and is wholly unnecessary if policy iteration is used.

Policy iteration is generally regarded as a faster algorithm than value iteration [145]. This can be broken down into two parts. First, unless ϵ is set such that value iteration terminates the moment the optimal policy is found, policy iteration will run for fewer iterations overall. Second, policy iteration is linear because it does not rely on a max operator in Equation 2.5. This results in an overall faster calculation of U. Of course, policy iteration also relies on a second calculation for policy improvement. However, this is (1) often negligible and (2) can be mitigated with *modified policy iteration* where policy evaluation runs k times rather than just once.

Part II

Research Question 1

Chapter 3

Generating Level Segments

> Only Odysseus time and time again turned craning toward the sun, impatient for the day's end, for the open sea.
>
> Homer, *The Odyssey*

This Chapter is based on the paper, "Gram-elites: n-gram based quality-diversity search" [16].[1,2] The authors are myself, Alejandro Hervella, and Seth Cooper.

The goal of this Chapter is to answer **RQ1.1**: How can usable level segments be generated within an organized structure applicable to dynamic difficulty adjustment? The answer to this question depends on the requirements for a level segment. For starters, recall the limitation presented in Section 1.4 that the generator needs only to generate levels for a 2D grid-based game. Given that limitation, two requirements for any generated level segment are relevant to this Chapter.

The first requirement is that every level segment shown to the player must be *completable*. A level segment is completable if an agent can beat it. Some work has shown that there is value in purposefully generating levels that are not completable [36]. However, those contexts—understanding what makes a level not completable, creating training data, generating levels that are not completable without a particular mechanic—are, for the most part, not relevant to this dissertation, which ultimately intends to use generated level segments as part of a level assembly process for DDA.

The second requirement is that generated level segments should look and feel like a level

[1] Code available on GitHub: `https://github.com/bi3mer/GramElites`

[2] Data generated for this Chapter available on GitHub: `https://github.com/bi3mer/GramElitesData`

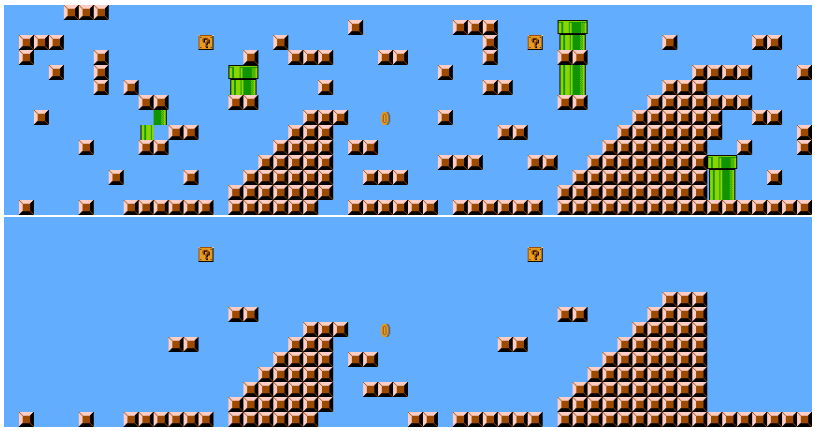


Figure 3.1: Example of two levels that are completable. They also share a common structure and a similar solution path. However, the top level looks nothing like you would expect a *Mario* level to look (chaotic, disorganized, etc.), whereas the bottom has the expected simplicity of a *Mario* level. The chaotic level also has a broken pipe on the left side.

in the target game—which is to say that a level segment generated for *Mario* should look like it belongs in *Mario*. If, for example, there were a bunch of random solid blocks all across the level, then that would not look like a *Mario* level; see Figure 3.1. This Figure also shows another potential problem with generated *Mario* levels: broken pipes. The general term is a *broken structure*. A *structure* is a multi-tile asset that follows a set pattern. A broken structure, therefore, is a structure that fails to meet the prescribed pattern of the target structure. The pipe in *Mario* is always two columns wide and at least two rows tall, but it can be more. The top row differs from the rows below in that they have a rim for the pipe. A pipe that does not have these characteristics is a broken structure. Unbroken structures are essential because any broken structure will stick out like a sore thumb to any player familiar with the game.

Now that the second requirement is defined, the question is: How can a level segment be classified computationally to meet the second requirement? This means that the algorithm needs to be able to classify, without human input, whether a level looks and feels like a level in the target game. Further, it needs to determine whether that level contains no broken in-game structures. The approach to solve this comes from the field of PCGML [175]. PCGML is a branch of PCG that uses Machine Learning (ML) algorithms trained on existing game content to generate new content in a similar style. N-grams, a ML technique (see Section 2.4), have been used to generate *Mario* levels [43]. Given a large enough N, an n-gram will produce output similar to its input. An excellent property of the n-gram is that it is not one-way, meaning it can both generate a level and assess

the likelihood of a level being generated. An n-gram can declare whether or not a level segment is *generable*. If a level is not generable, then the n-gram could not generate that level. This means that a non-generable level has characteristics that are not similar to those of the target game. Therefore, n-gram generability is used to assess style and determine whether or not a broken structure exists.

The two requirements for a level segment are:

1. Completable - an agent can beat the level.

2. Generable - the level is n-gram generable.

If a level passes both requirements, then the level is *usable*.

With level segment requirements defined, the next step is to determine how level segments are generated. Again, we will start with the requirements. The first requirement is that the generator can generate usable level segments. The second is that the generator needs to generate multiple segments. This second requirement naturally points to a GA, which generates a population of solutions. However, these solutions will point towards a single optimal point (as defined by the fitness function), meaning the final population is unlikely to be diverse. Diversity is key in this domain because players should not have to repeatedly play similar level segments. This is remedied by using a QD algorithm, specifically MAP-Elites [67, 129]; see Section 2.5.1.

The problem with the approach as described so far is that MAP-Elites will generate levels that are not n-gram generable with standard genetic operators [69, 203].[3] This causes wasted computation. The solution presented in this Chapter is n-gram genetic operators. N-grams are used to modify a given solution for both mutation and crossover. As a result, a new solution will always be generable by an n-gram. The combination of n-gram operators and MAP-Elites is called *Gram-Elites*.

Gram-Elites was tested with three tile-based games: *Super Mario Bros.* [131], *Kid Icarus* [132], and *DungeonGrams*—described in Section 3.6. Gram-Elites was compared to two variants. The first was MAP-Elites with standard operators (crossover and mutation). The second variant placed segments that were generated by an n-gram into the MAP-Elites grid. For all three games, Gram-Elites found more usable segments in fewer iterations.

[3]The top level of Figure 3.1 is a good example of this.

3.1 Related Work

The work that is most relevant to this Chapter has already been discussed at the end of Section 2.4, and it showed how n-grams can generate *Mario* levels based on level slices [43]. They found that $n = 3$ produced results aligned with the style of a *Mario* level while still appearing novel. In this Chapter, n-grams generate level segments, assess levels for generability, and modify levels with genetic operators.

The other major component of this Chapter is Procedural Content Generation Through Quality Diversity (PCGQD). Gravina et al. introduced the topic and broke down different QD approaches [67]. The core element of a QD algorithm is to "search for a set of good solutions which cover a space as defined by behavior metrics." In effect, find quality through diversity. Another term for QD is an *illumination algorithm*. The goal of an illumination algorithm, according to Mouret and Clune, is to illuminate "the fitness potential of each area of the feature space, including tradeoffs between performance and the features of interest" by finding the "highest performing solution for each cell in the N-dimensional feature space" [129].

MAP-Elites [129] is a QD algorithm, but when used for PCG, it is referred to as a PCGQD algorithm. Withington showed that MAP-Elites can generate Mario levels by using a binary representation for whether a block exists or not [203]. The generated segments are completable, but they do not look or feel like a Mario level. Khalifa et al. [100] address the problem of structure by using Constrained MAP-Elites [101]. This builds off of FI-2Pop [102] to have valid and invalid levels inside of every cell in the MAP-Elites grid. A level is evolved to be completable by an agent in the infeasible population. In a feasible population, levels are evolved to be as simple as possible. This is another way of assessing structure that differs from our approach.

While this Chapter uses MAP-Elites, MAP-Elites is not the only PCGQD method as shown by Gravina's survey. One example comes from Beukman et al. [12]. In their work, they used neuroevolution [170] and novelty search [107] to find diverse video game levels. While they did not compare to MAP-Elites, their results for maze generation and level generation for *Mario* were promising.

Fontaine et al. [54] showed how a quality diversity search can work with a generative adversarial network (GAN) [64]. Instead of evolving levels, they evolve vectors as input into the latent space of the GAN. The GAN's objective is to output levels that match the structure of the target game.

This Chapter examines how n-grams can improve MAP-Elites for level generation, but

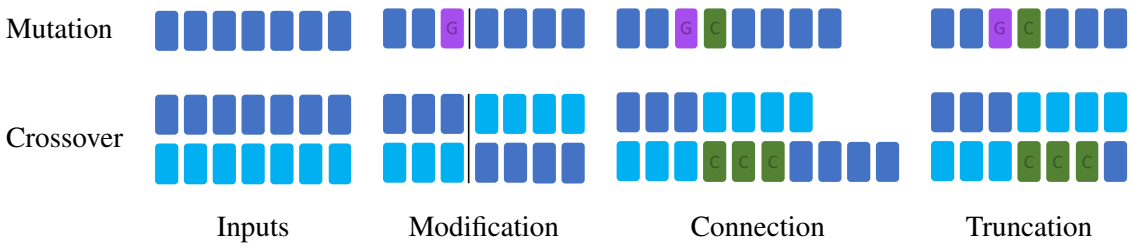


Figure 3.2: Visualization of n-gram genetic operators. Both operators build a connection (green Cs) to the other side to maintain a full segment with no bad n-grams. The mutation operator removes one slice and uses the n-gram to generate a replacement (purple G) before the connection is built.

there is another work that is similar. Lo et al. evolved musical sequences [114] by using a complex fitness function that included n-grams to assess the likelihood of a given sequence. In the context of music, it can be beneficial to evolve a sequence that is not generable by an n-gram.

3.2 N-Gram Operators

Standard genetic operations can create sequences that are not n-gram generable. N-gram genetic operators were developed to address this for mutation and crossover; see Figure 3.2. Both n-gram genetic operations are designed such that given an input sequence that is generable by the n-gram, the output sequence will also be n-gram generable.

Both operators rely on a concept called *connection*. A connection is built between the two sequences to combine them. This is built using the last $n-1$ slices of the starting sequence (start prior) and the first $n-1$ slices of the ending sequence (end prior). The connection is built using a Breadth-First Search (BFS) on the n-gram. It starts from the start prior and searches for a path to the end prior. Note that for the approach to work, such paths must exist between priors in the n-gram and thus rely on using an n-gram with at least one path between the start prior and end prior. A path is guaranteed to exist if the n-gram is *strongly-connected*. An n-gram is strongly connected if and only if every prior has a path to every other prior in the n-gram—see Section 2.4.1 for more on strongly connected n-grams. BFS results in a potentially empty sequence of slices that connects the two original sequences. With *connection*, the operators n-gram mutation and n-gram crossover are possible.

- *n-gram mutation* chooses a random point in a sequence. The random point is bounded by the requirement that there must be $n-1$ slices on either side of it. The slice at this point is removed. A new output is generated using the last $n-1$ slices of the sequence to the left of

the point. The new output updates the start prior. The start prior is used to connect the two sequences around the random point. See Figure 3.2 (top).

- *n-gram crossover* receives two parent segments and finds a crossover point. This point has the exact requirements as mutation. The point is selected randomly, but there must be $n-1$ slices on either side of the point for both parent segments. Crossover is run with both segments with only one difference: an n-gram connection is built between the two segments. See Figure 3.2 (bottom).

Both of the n-gram operators can increase the size of a sequence. The case for increasing the size of a sequence is shown in Figure 3.2. To enforce a fixed sequence length, slices past the maximum length are truncated for both operators.

3.3 Method

Three games were used to evaluate Gram-Elites: *Mario*, *Kid Icarus* (shortened to *Icarus* for the remainder of this dissertation), and *DungeonGrams*. *Mario* and *Icarus* are platformers. *DungeonGrams* is a roguelike developed as an additional challenge for the level generator—*DungeonGrams* is described later in Section 3.6.

3.3.1 Fitness Function

All three games use the same two-part fitness function ($f(s)$), which MAP-Elites aims to minimize. A segment s is *usable* if $f(s) = 0$.

$$f(s) = B(s) + 1 - C(s)$$

- $B(s) \in \mathbb{N}_0$ represents the number of bad-n-grams in a level. A bad n-gram results from an unseen prior or an unknown output given a valid prior.
- $C(s) \in [0, 1]$ represents the percent that an agent can complete the level segment s.

The agent in $C(s)$ receives a modified version of the generated level segment s. The modification is to add padding to both sides of the level [149]. This allows for an easily definable start and end point for the fitness function.

3.3.2 Generators

Three generators were tested:

- *ME-NGO* - MAP-Elites with n-gram operators (i.e., Gram-Elites).
- *ME-SO* - MAP-Elites with standard operators.
- *NG* - n-gram generation with a MAP-Elites grid.

All three rely on placing level segments into the MAP-Elites grid based on the level segment's BCs, see Section 2.5.1 for more on how placement works. Each cell in the grid was configured to allow for a maximum of four elites.

ME-NGO and ME-SO start with population generation. In this phase, an n-gram is run to generate level segments of a target length, and the result is placed into the grid following the described procedure above. This runs for 500 iterations. After, ME-NGO and ME-SO run with their respective genetic operators for $50,000$ iterations. The selection strategy used by both is random for selecting the grid cell and the elite inside the cell. Crossover was always performed. Both child sequences of crossover had a 2% chance of mutation.

NG is ME-NGO and ME-SO, except its configuration is to run 0 iterations of genetic operators, and it runs population generation for $55,000$ iterations, generating a random segment with an n-gram every iteration. Also note that for both NG and ME-NGO, $\forall s : B(s) = 0$ is true for the fitness calculation.

3.3.3 A Note on Evaluating Quality-Diversity Algorithms

A standard metric used to assess MAP-Elites is the QD-Score [140]. This score is the sum of the highest fitness values found in every cell in the grid. Its goal is to quantify both quality and diversity found after a MAP-Elites finishies running. QD score was not used for two reasons. First, the goal is to minimize $f(s)$, not maximize, which means a large QD would be bad. Second, since QD-score uses $f(s)$, quantifying an empty cell as the maximum number of bad n-grams plus one could be overly harsh. The combination of these two facts indicates that a QD-score would be misleading; thus, it was not used.

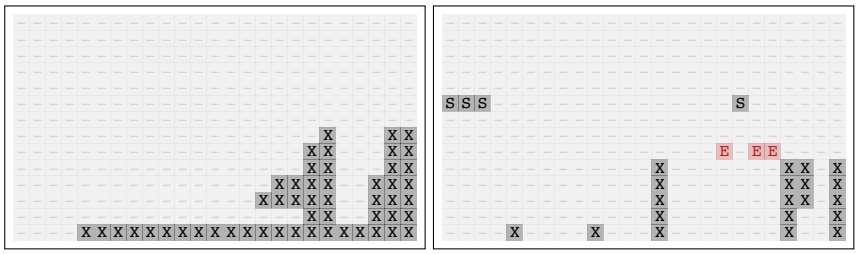


Figure 3.3: Two example level segments generated by Gram-Elites for *Mario*. Grey 'X' blocks are solid blocks the player can run on. The grey 'S' represents a solid brick block. The red 'E' represents an enemy. There is no coin in either of these level segments, but a coin is represented by a yellow 'o'.

3.4 Case Study 1: *Mario*

Mario [131] is a real-time horizontal platformer. Figure 3.3 shows two sample levels generated in this case study by Gram-Elites.[4] The player starts at the left-most point of the level and has to traverse to the right, which is why it is called a horizontal platformer. The player can lose by falling through gaps or running into enemies, but the player can jump on top of enemies without losing, and the enemy is destroyed in the process.

3.4.1 Agent

Rather than create a working version of the game, the Mario AI Framework [185] and Summerville's "A star" search (A*) agent [174] were tested. The *Mario* AI Framework implemented *Mario* in Java,[5] with a working implementation of A* [78] builtin that returned additional stats like the number of jumps used to beat the level. Summerville's A* agent was written in Python[6] and only returned a path if a level was completable, with no additional features.

We used Summerville's A* agent because it was faster and easier to work with. It was faster because it simulated fewer elements than the Java version. These elements, while interesting, were not necessary for this specific use case. It was easier to work with because it was written in Python, the language the codebase for Gram-Elites is written in. This allowed for quicker iteration while testing.

[4] Figures of this kind were generated with level2image (citation not yet available). The code is available on GitHub: `https://github.com/crowdgames/level2image`

[5] `https://github.com/amidos2006/Mario-AI-Framework`

[6] `https://github.com/TheVGLC/TheVGLC`

Summerville's A* agent was partially modified. The original version of the code was a traditional implementation of A* where an empty path was returned if a path did not exist from the start to the end of an input level. The code was modified to return the percentage that the agent was able to complete a given level. The change made the code track the furthest point reached by the agent. If the agent reached the end of the level, the search was terminated and returned 1.0. Otherwise, the variable that tracked the furthest point reached by the agent was divided by the total length of the level (number of columns), and the result was returned.

3.4.2 Configuration

- *N-gram*: The n-gram for *Mario* uses the *Super Mario Bros.* levels in the Video Game Level Corpus [176] that are not underground, do not have moving platforms (the grid representation does not capture the behavior well), and do not have springs (they are not present in the processed map files). All input levels are broken into vertical level slices, or columns; see Figure 2.2. The n-gram was a tri-gram based on the work from Dahlskog et al. [43].

- *Generated segment size*: 25 columns.

- *Resolution*: A resolution of 40 was used for each BCin the MAP-Elites grid—Figure 3.6 shows this. Meaning there are 40 bins for each behavior. With two dimensions, this results in 40^2 or $1,600$ bins that could be populated with elites.

- *Padding*: A padding of two columns was added to the left and right sides of the level segment. The columns were empty, besides a solid block at the bottom.

- *Behavioral Characteristics*:

  - Linearity [162] is calculated by finding the minimum height for every column and finding the line of best fit. For every minimum height, the absolute value of the difference between it and the line of best fit (calculated based on those found heights) is calculated. The result is divided by the maximum linearity for a level segment to give a percentage.
  - Leniency [162] is calculated based on whether a column contains an enemy $(+1/2)$ or contains a gap $(+1/2)$ for a max score of 1; for a score of 1, a column must contain an enemy and a gap, not two enemies. This value is divided by the number of columns in the level to get a percentage.

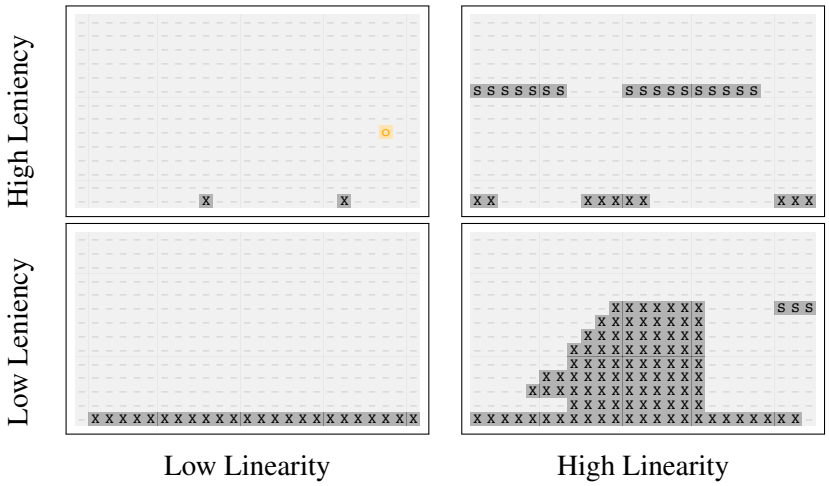


Figure 3.4: Sample levels generated by Gram-Elites for *Mario* to show levels of high and low behavioral characteristics.

3.4.3 Evaluation

Segments generated by Gram-Elites, organized by BCs, can be seen in Figure 3.4. The bottom row—low leniency—shows two levels that are not challenging to a player familiar with *Mario*. Both levels have no enemies, and the gaps are tiny, requiring a small jump from the player to clear. The level with high linearity does require more jumps, but there are no stakes in the sense that if the player messes up, they will not lose an in-game life. Conversely, the top row—high leniency—had more challenging levels. Both require precise control from the player to make the jumps. The top-left level, which was low linearity and high leniency, only required two jumps, but the player's input needs to be near perfect to beat the level segment. The segment with high linearity and high leniency has a reasonably linear solution path, requiring only two jumps. This occurs because the linearity calculation is not based on the path but on the lowest block in each column—this is discussed more in Section 3.8.

Figure 3.5 shows how many levels were found by each algorithm (ME-NGO, ME-SO, and NG). As seen across the one hundred runs, the results are consistent based on the 95% confidence interval. NG performed worst, finding just over ≈ 250 usable level segments. ME-SO found more levels than NG after less than $5,000$ iterations. ME-SO reached its peak usable levels found around $30,000$ iterations. ME-SO found a total of ≈ 350 usable level segments. ME-NGO outperformed ME-SO by $5,000$ iterations completed and stopped consistently finding levels at around $20,000$ iterations. ME-NGO found in total ≈ 400 usable level segments.

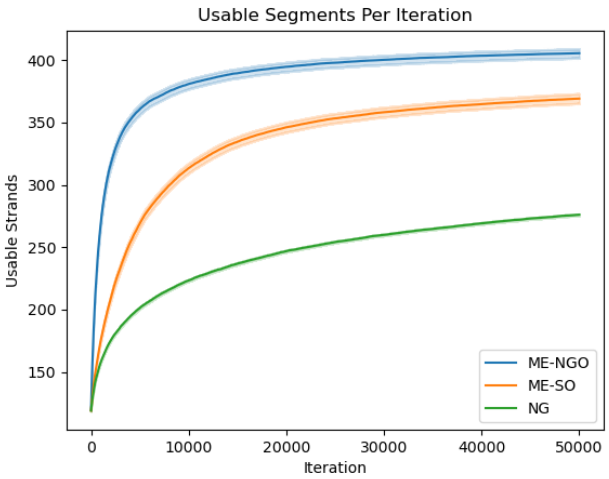
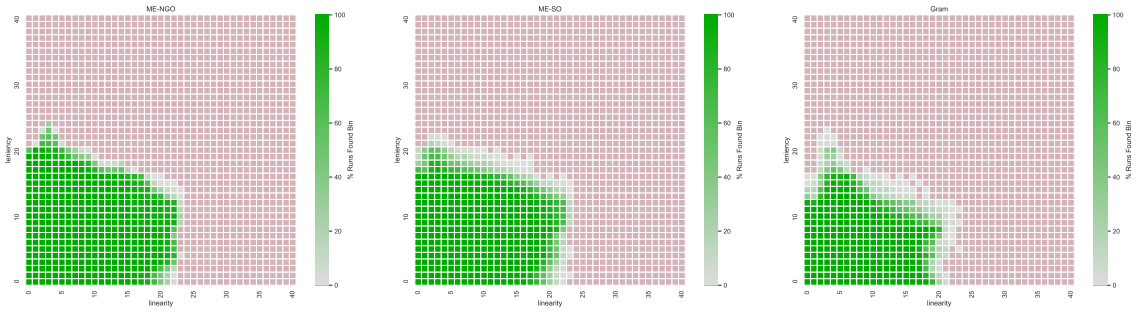


Figure 3.5: Plots of usable segments per iteration for *Mario*. Plots show the average and 95% confidence interval for 100 runs.



ME-NGO ME-SO NG

Figure 3.6: MAP-Elites grid for all three algorithms and games over 100 runs for *Mario*. The value is the percentage of runs that found the bin.

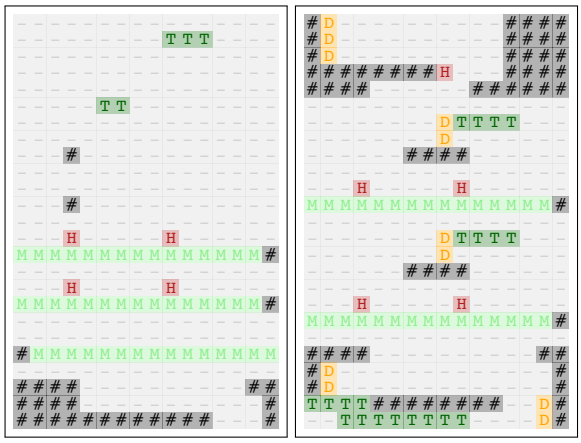


Figure 3.7: Two example level segments generated by Gram-Elites for *Icarus*. The grey '#' represents a solid block. The green 'T' represents a block that the player can jump through, but is solid when they land on top of it. The green 'M' represents a path for a moving platform of four blocks that the player can jump through and land on top of. The yellow 'D' is a door that the player can use to exit the level. The red 'h' is a hazardous block that will kill the player if they touch it.

Figure 3.6 shows the MAP-Elites grid filled by all three algorithms on average across all one hundred runs. These are difficult to decipher at a glance, but based on the discussion of Figure 3.5, it is simple to say that ME-NGO filled in more space. The easiest place to see this is linearity 8 and leniency 20. This shows that NG filled a few cells less consistently in that area. It also shows that ME-SO and ME-NGO consistently filled in that area. However, ME-NGO was able to go higher into the grid inconsistently, whereas ME-NGO could only consistently reach leniency 24. Another example is towards the border at high linearity where Gram-Elites, again, better filled in the grid.

3.5 Case Study 2: *Kid Icarus*

Kid Icarus [132] is a real-time vertical platformer where the player starts at the bottom of a level and has to reach the top. Figure 3.7 shows two example levels generated by Gram-Elites. This figure includes an example of how moving platforms can be represented in a grid of tiles with the 'M' block, representing a path for the moving platform to follow. The player can lose by running into a hazard or falling off the game's screen.

3.5.1 Agent

The Summerville A* was used to test completability. The reasons to use the Summervvile agent are the same as presented above for *Mario*. However, there is no equivalent Mario AI Framework [185] for *Kid Icarus*. Therefore, the options were to use the Summerville A* agent or create a new version of *Kid Icarus*. The time commitment for the former was much more reasonable, and there was no additional benefit to making a clone.

Two modifications had to be made to Summerville's agent for *Icarus*. First, it was modified to solve a vertical level rather than horizontal. This was accomplished by changing where the goal was and modifying the tracking variable for the furthest point reached to now track the highest y-coordinate the agent reached. Second, new movement patterns had to be added to the agent. Specifically, in *Icarus*, the player can loop from one side of a level to the other. If the player goes off the screen on the right side, they go to the left side at the same height. Similarly, if they go offscreen on the left side, they go to the right. With these two changes, the agent could approximate how completable a *Kid Icarus* level was.

3.5.2 Configuration

- *N-gram*: The n-gram for *Icarus* uses all the *Kid Icarus* levels in the Video Game Level Corpus [176]; there are 6 in total. *Icarus* uses rows for level slices and n-gram with $n = 2$ for more variety based on past work [37]; this is due to the number of and size of input levels being far more limited than *Mario*, which leads to an n as small as three resulting in increased memorization.

- *Generated segment size*: 25 columns.

- *Resolution*: 35 for each behavior, with a maximum of 0.5 for each dimension.[7]

- *Padding*: A padding of two rows was added to the bottom and top of the level segment. The bottom row was filled with solid blocks. The other three rows were empty.

- *Behavioral Characteristics*:

[7] This was based on testing where it was found that the expressive range of the generator was more limited for *Kid Icarus*, see Figure 3.10 to see a visual of this.

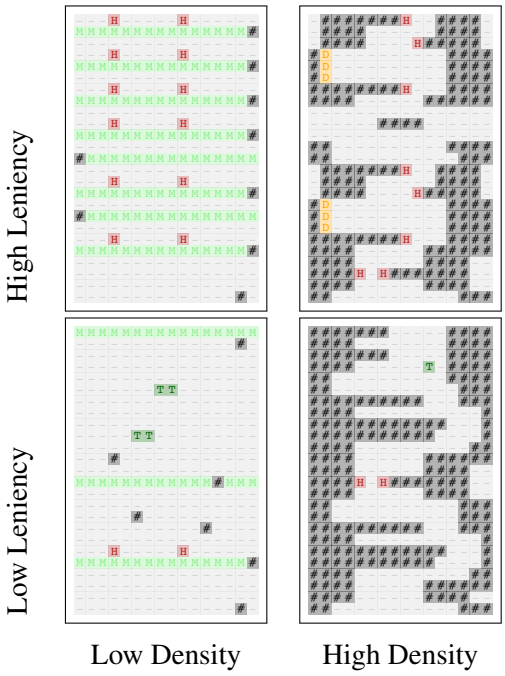


Figure 3.8: Sample levels generated by Gram-Elites for *Icarus* to show levels of high and low behavioral characteristics.

- Density is the number of solid tiles divided by the total number of tiles; density is the percent of solid tiles in the levels. 'T' blocks were counted as solid blocks, but 'M' was not.
- Leniency [162] is similar to the heuristic used in Case Study 1. Every row is evaluated for containing a door ($+1/3$), a moving platform ($+1/3$), and a hazard ($+1/3$) for a max score of 1. If there is more than one of these, this does not affect the final count for each row. The final leniency is divided by the number of rows in the level to give a percentage.

3.5.3 Evaluation

Segments generated by Gram-Elites, organized by BCs, can be seen in Figure 3.8. The lower left level was the easiest to beat in the previous case study, but this is not true for *Icarus*. It is a high stakes level with little room for error, because if the player misses a platform, then they will lose. Lower density results in levels with fewer platforms for the player to jump to. The level uses moving platforms so that the level is completable, but has low density. As a reminder, the 'M' block represents not only the platform for traversing up the level but also the path of the moving platform,

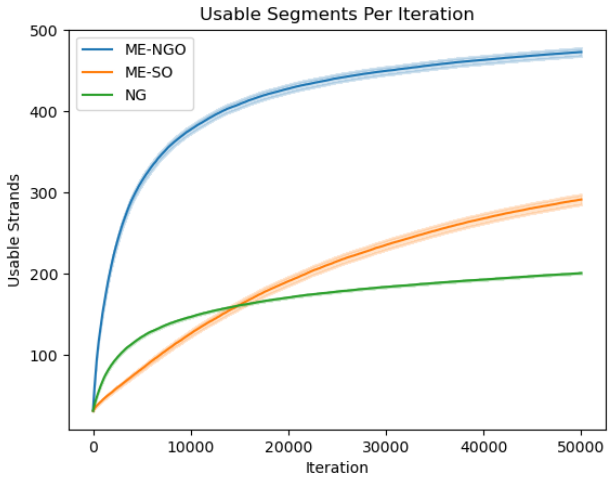


Figure 3.9: Plots of usable segments per iteration for *Icarus*. Plots show the average and 95% confidence interval for 100 runs.

which is why the 'M' blocks do not count towards density. The level with high leniency and low density did not result in a more complex level because there are many more moving platforms for the player. Further, this level shows that repeating patterns are sometimes used to reach a target BC.

The two levels with high density show a different kind of difficulty. The player is no longer at risk of falling off the screen, but hazards are no longer avoidable for the player. This means that to progress upward, the player has no choice but to go closer to hazards. This requires more fine-grained control from the player. As a result, higher density can also make fairly difficult levels, especially when combined with high leniency.

Figure 3.9 shows how many levels were found by each algorithm for *Icarus*. The results are similar those of Case Study 1 but more extreme. NG starts by outperforming ME-SO but is overtaken by about $15,000$ iteration, and it peters out at about 200 usable level segments found. ME-SO found ≈ 300 usable level segments. ME-SO appears to be still finding usable level segments at the end of $50,000$ iterations, but it took a while to find each new segment. ME-NGO found more levels than NG and ME-SO before reaching $5,000$ iterations. It stops at ≈ 475 usable level segments found.

Figure 3.10 shows the resulting MAP-Elites grids when averaged across all one hundred runs. Based on the results from Figure 3.9, it is unsurprising to see that these graphs are much easier to understand at a glance than the corresponding graphs in Case Study 1. ME-NGO was able to fill in all the space covered by the other two algorithms and more. ME-SO outperformed NG, and NG

ME-NGO ME-SO NG

Figure 3.10: MAP-Elites grid for all three algorithms and games over 100 runs for *Icarus*. The value is the percentage of runs that found the bin.

struggled the most. Interestingly, it was unable to find level segments with lower density. The same is partially true for ME-SO, but ME-NGO had no trouble. It appears that it was impossible to make a level with 0 density. Similarly, levels with high levels of leniency appear impossible to generate given the input level segments from the VGLC.

3.6 Case Study 3: *DungeonGrams*

DungeonGrams is a small, turn-based rogue-lite game where the player's goal is to make it out of the dungeon. Figure 3.11 shows an example level segment generated by Gram-Elites, which has been imported into the web version of the game.[8] [9].

The game starts with the player at the top left of a level, and they have to make it through a portal that is always at the bottom right. If there are keys in the level, though, the portal is closed until the player has collected every key. The player can collect a key by stepping on the tile where the key is. Once the portal is open, the player can step onto the portal's tile to win.

The game is made more difficult by a stamina mechanic. The player starts with 40 stamina. Stamina is reduced by one when the player moves, runs into a wall, or pauses for one turn. The player loses if the player runs out of stamina before stepping through the open portal. If the player steps through the open portal at their last stamina point, the tie is broken in the player's favor, and they win. Stamina can be replenished by food, increasing the player's stamina by 25, up to a max of 40.

[8] Playable at: `https://bi3mer.github.io/DungeonGrams-TS`
[9] Code available on GitHub: `https://github.com/bi3mer/DungeonGrams-TS`

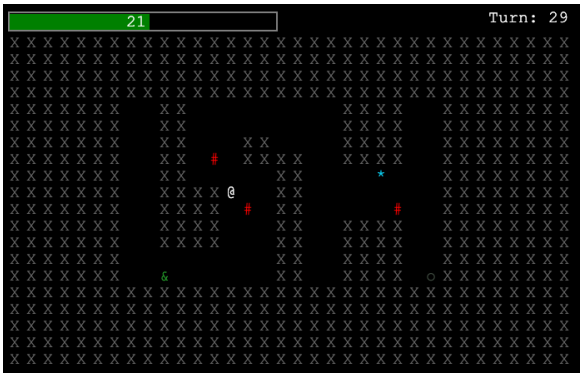


Figure 3.11: Example screenshot from the web version of *DungeonGrams*. The white '@' represents the player. A red '#' represents an enemy. A blue '*' represents a key that must be collected before the portal opens. The portal is closed because the player has not collected every key in the level. The portal is the dark grey 'o' at the bottom right. When the player has collected every key, the portal turns into a green 'O'. The green '&' represents food and it will return stamina to the player. The green bar at the top left represents how much stamina the player currently has.

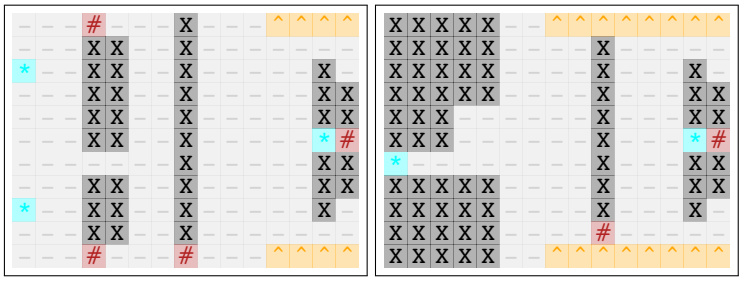


Figure 3.12: Two example level segments generated by Gram-Elites for *DungeonGrams*.

In addition to stamina, there are also enemies. If an enemy and the player come into contact, the player loses. Enemies can move every third turn (turns 3, 6, 9, etc.). On their turn, they will chase the player if and only if the Euclidean distance from them to the player is less than or equal to three and the Euclidean distance from their starting point is less than or equal to 3. If either condition fails, the enemy moves back to their starting position. Two sample levels generated by Gram-Elites are shown in Figure 3.12.

3.6.1 Agent

The game was implemented in Python, and it includes an A* agent to test if a level is beatable.[10] The state space is large enough that a complete search becomes too time-consuming. For quicker evaluation, the agent was made greedy and given a search space limit based on the size of the level being solved. Completability is the following:

$$C(s) = \begin{cases} 1, & \text{Agent won} \\ 0.9\left(\frac{a_s}{l_s} + \frac{a_x}{l_w}\right), & \text{Otherwise} \end{cases}$$

- 0.9 - a scaling factor that distinguishes between when the agent does beat the level segment s and does not.[11]
- a_s - the number of switches the agent switched.
- a_x - the maximum x-coordinate the agent reached.
- l_s - the number of switches in the level segment s.
- l_x - the maximum width or furthest x coordinate in the level.

3.6.2 Configuration

- *N-Gram*: 44 small levels were manually built for *DungeonGrams* with 11 rows each. These levels were intentionally designed to have repeating patterns that an n-gram model could use. *DungeonGrams* used a 3-gram with level slices as columns.
- *Generated Segment Size*: 15.

[10] Code available on GitHub: `https://github.com/crowdgames/dungeongrams`
[11] An agent may reach all the way to the right of a level and hit all the switches but not beat the level.

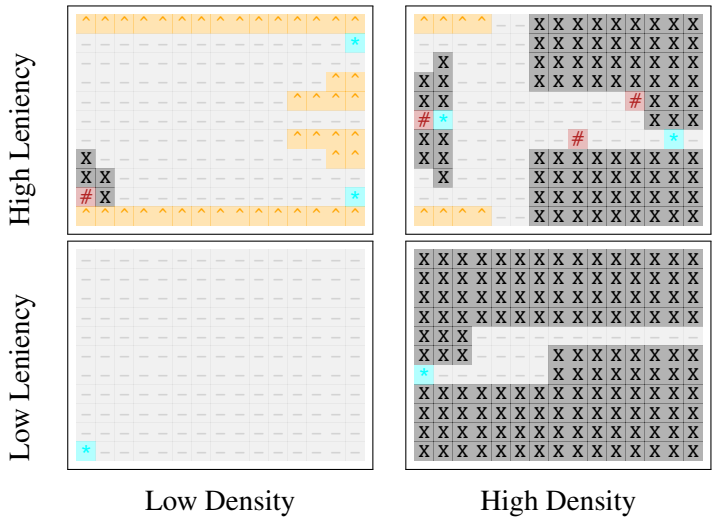


Figure 3.13: Sample levels generated by Gram-Elites for *DungeonGrams* to show levels of high and low behavioral characteristics.

- *Resolution*: 20 for each BC. However, the max leniency is set to 0.5 to tighten the search space in that dimension.
- *Padding*: Two empty columns on both sides. The top left tile of the left-most column is modified to indicate to the game where the player starts. The bottom right tile of the right-most column is modified to include a portal.
- *Behavioral Characteristics*:
  - Density is used again and follows the same description from Case Study 2. Spikes are not counted as solids in the function.
  - Leniency is similar to the other two case studies. Every column is evaluated for containing an enemy $(+1/3)$, a spike $(+1/3)$, and a switch $(+1/3)$ for a max score of 1. Note that two enemies in the same column would only add $1/3$ and not $2/3$ to the score. The final leniency is divided by the number of columns in the level to give a percentage.

3.6.3 Evaluation

Figure 3.13 shows levels generated by Gram-Elites separated by BCs. First, looking at the level that corresponds with low density and low leniency, it can be seen that the resulting level has no challenge for the player other than to hit the switch. The level with low density and high leniency

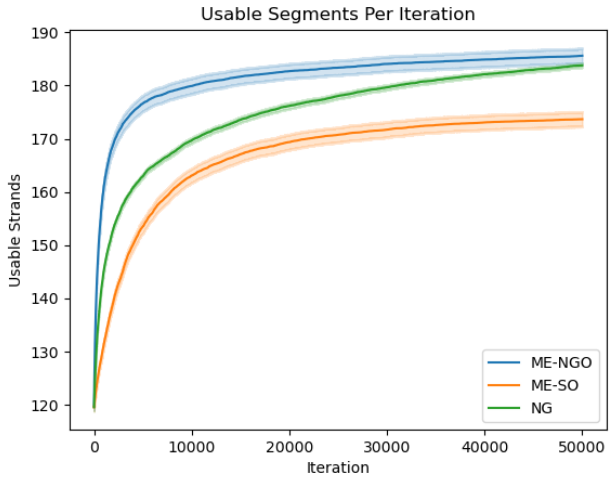


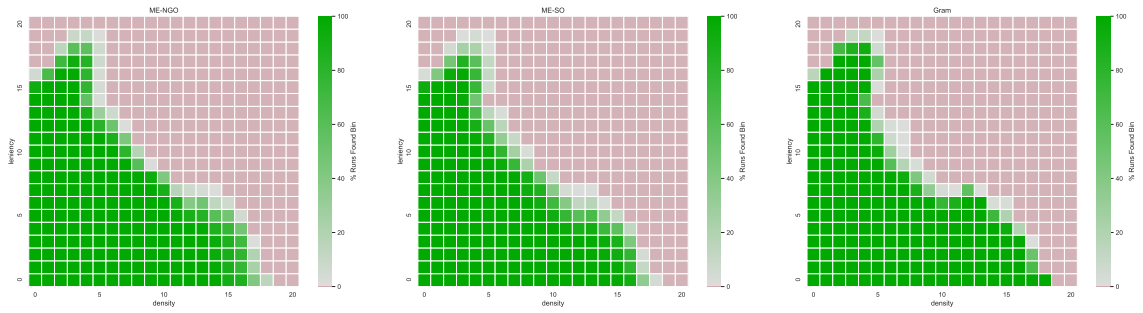
Figure 3.14: Plots of usable segments per iteration for *DungeonGrams*. Plots show the average and 95% confidence interval for 100 runs.

has two switches, one enemy, and many spikes. This illustrates a downside of this approach seen in the other two Case Studies: BCs can be abused. Spikes, while a hazard, do not make the game much more difficult for the player, especially in a turn-based game. They are essentially the same as a solid block. As a result, a better approach would have likely been to either not count spikes at all for leniency or downgrade them such that they are not considered equal to enemies and switches.

Moving on to the level segment that corresponds with high density and low leniency, it can be seen that the player has no choice but to hit the switch if they want to complete the level. After the switch, the path is clear on how to beat the level. Moving on to the last level, top-right, it can be seen that when combining both BCs, the levels can become pretty challenging. Precise timing is needed to switch the left-most switch, and experience with the enemy AI is required to navigate the narrow passage guarded by the two enemies correctly.

Figure 3.14 shows the number of usable level segments generated by all three algorithms for *DungeonGrams*. Of all three case studies, this is the only one where ME-SO is the worst-performing approach. ME-NGO and NG are both better performing from the very start. ME-SO found ≈ 170 usable level segments. NG is slower to find levels than ME-NGO but appears to find levels at the end of $50,000$ iterations. NG found ≈ 180 usable level segments. ME-NGO found ≈ 185 usable level segments.

Compared to the other two case studies, all three algorithms found fewer usable level segments. The reason for this is that this case study used a smaller resolution for the MAP-Elites



ME-NGO ME-SO NG

Figure 3.15: MAP-Elites grid for all three algorithms and games over 100 runs for *DungeonGrams*. The value is the percentage of runs that found the bin.

grid.

Figure 3.15 shows the resulting MAP-Elites grids when averaged across all one hundred runs.[12] The simplest way to find the differences between the three graphs is to look towards the front as all three algorithms try and fail to reach the top-right of the grid. NG is most consistent in finding a high density, low leniency at the bottom right. NG, though, does worse towards the middle of the front, at around density 10. This is the same problem exhibited by ME-SO, whereas ME-NGO can do a slightly better job on average.

3.7 Limitations

- Grid-based - Gram-Elites relies on n-grams and will, therefore, not work on games that are not grid-based. Further, it relies on breaking a level into level slices for the n-gram. While there has been work that generated levels tile by tile—snaking up and down along columns [173]—this approach would likely not work if attempted with Gram-Elites because long-term dependencies are not well-captured. An example of a long-term dependency is a gap that grows too large and becomes impossible to jump over.

- BCs - As noted in each Case Study, BCs can be abused to find levels that do not meet what the designer intended. The solution is to conduct small runs of Gram-Elites and test until satisfied with the results produced by the BCs.[13]

[12] It may be helpful to compare Figures 3.10 and 3.15 to visualize how resolution affects the number of usable level segments generated.

[13] I am not aware of any better solutions to this problem as of writing this dissertation.

- Training data - Gram-Elites is a PCG through QD approach [67], but it relies on PCGML [175]. PCGML has a fundamental tension built into it: the system needs good training data to generate good levels [94]. Producing that training data is time-consuming, and it raises the question, "Why generate levels when the built training data could be used?" This Chapter and the methods presented do not address this question, and it is left up to the reader to decide whether making levels to generate new levels is worthwhile.

- N-gram Generability - Generability restricts the generative space. Some levels that look and feel like a valid level segment to a human are not generable by an n-gram. Therefore, the solution is to add more training data for the n-gram. However, as noted in the bullet point above on training data, this is far from an ideal solution.

- Agent - If developers want to use Gram-Elites for their games, they must create a completability agent. For some games, this is simple. For others, it is not. For example, *DungeonGrams* is a simple game; therefore, it is reasonable to assume that making an agent for it would also be simple. However, as noted in Section 3.6, the A* agent ended up running slowly due to the size of the state space.[14] This same problem is true for platformers as the agent used in this Chapter simplified the search space by not simulating enemy movement. Therefore, designers must think deeply about the agent if they want to use Gram-Elites.

- Time - The speed of the agent to solve a level segment and the number of iterations to run are the two significant factors in determining how quickly Gram-Elites runs. While the method is not slow to find levels, it is not fast enough that I could recommend using it in an online context where levels are generated moments before the player plays. Instead, Gram-Elites should be seen as an offline PLG method that generates level segments before the player has opened the game. These levels could be loaded in from a server or packaged with the game.

3.8 Unexplored Areas of Future Work

- For Gram-Elites to work, there must be a path from the start prior to the end prior. In the case of *Mario* and *DungeonGrams*, both n-grams are strongly connected—see Section 2.4.1. In the case of *Icarus*, though, the n-gram had "dead starts" or priors with no incoming edges. Because the current implementation chooses mutation and crossover points from the middle,

[14] Solution can be seen in the agent's code: `https://github.com/crowdgames/dungeongrams/blob/main/dungeongrams.py#L460`

these dead starts can only appear at the start of any segment. This, though, is problematic since the start prior is limited. If *Icarus* were used in Part III, then the work presented would be redone with a strongly connected n-gram. Further, it would be interesting to compare the results of both n-grams.

- As noted before in the Section 3.7, the size of n n-gram affects the results of the level. An extension of Gram-Elites would be a method to find the best n to produce recognizable structures without memorization. It could also be interesting to characterize the performance of ME-NGO relative to ME-SO based on structural properties of the n-gram, such as the number of outputs for each prior.

- The linearity heuristic for *Mario* is based on column heights. Figure 3.4 shows a level with high leniency and high linearity. However, that level does not require the player to jump up and down from the platforms. The solution to improve this would be to run a naive path-finding algorithm on the level segment s and compute linearity with the changes in the player's height while solving, rather than column height. Re-running Gram-Elites with this improved version would likely give better levels, but could reduce the number of usable level segments found.

- One property of the current approach is that the n-gram operators do not take into account the likelihood of a particular sequence according to the n-gram, just that it can be generated. This may allow it to generate more unlikely "extreme" sequences in those areas of the cell, but also not reflect the distribution of slices in the training data. Incorporating the likelihood of a sequence being generated as a MAP-Elites behavior might help address this.

- The current output of Gram-Elites is an array of level slices. When put together, these level slices form a grid. Another approach that could give interesting results is to output a graph [35, 50, 88]. The main advantage of this change would be generality. The constraint of grid-based games could be removed. However, n-grams would have to be dropped in favor of Markov chains [145].[15]

- N-grams limit the generative space of the generator. They are necessary because they provide guarantees for the output level segments, but there are other ways. One approach is a local Markov model, which works tile by tile within a neighborhood of tiles [163, 164]. This

[15] N-grams are a specialized form of Markov chains for linear data, whereas a Markov chain can work with a graph-like structure.

model could be used to act as level repairers [37, 159] after standard genetic operators. This approach will still have the same problems as listed above—training data, neighborhood size and shape, etc.—but the local space could allow for a broader range of usable level segments. The other approach would be to use a backoff n-gram [91, 165]. A backoff n-gram starts with the highest order, say 3 for this example, but then backs off to 2 or even 1 if there is no output for a prior. This would allow for more exploration of the generative space, but the definition of what is generable and what is not would be less restricted, which could lead to the occasionally odd output that a human expert would have to filter.

3.9 Conclusion

This chapter answers **RQ1.1**: How can usable level segments be generated within an organized structure applicable to dynamic difficulty adjustment? The problem was broken down into several sub-questions, resulting in the constraint that a level segment has to be usable, which is to say that a level must be completable and n-gram generable. Further, the generator was required to create multiple level segments in one go. This problem was solved by creating Gram-Elites, an extension to MAP-Elites that uses n-gram population generation and n-gram operators for mutation and crossover.

Gram-Elites (ME-NGO) was tested against an n-gram generator (NG) and MAP-Elites with standard operators (ME-SO) on three games: *Mario*, *Icarus*, and *DungeonGrams*. For *Mario* and *Icarus*, NG found the least number of usable level segments. However, ME-SO was the worst performing for *DungeonGrams*. ME-NGO was the best performing for all three games, finding the most usable level segments and exploring the largest portion of the MAP-Elites grid.

Now that we can generate usable-level segments, the next part is to figure out how to combine them to form a full level. This can be done by turning the grid structure into a digraph with neighbors. However, it will be shown that the level segments cannot simply be concatenated together, and a better approach is required.

Chapter 4

Linking Level Segments

> It is better to know one book intimately
> than a hundred superficially.
>
> Donna Tartt, *The Secret History*

This Chapter is based on the paper, "On Linking Level Segments" [17].[1,2] The authors are myself and Seth Cooper.

This Chapter aims to answer **RQ1.2**: How can usable level segments be linked together to form levels? Before going into how this Chapter answers that question, it may be helpful to consider why this question is important.

Level segment generation is a promising area because it gives designers more fine-grained control. To better understand this claim, consider the task of building a *Mario* level with many jumps at the start, then a few enemies, then no jumps, and, finally, many jumps and many enemies. If a generator has to build the described level, then it has a large challenge due to the size of the search space. It can be more manageable to break the designer's requirements into parts by building smaller segments and combining them. These level segments, though, need to be placed together to form larger levels.

This Chapter will show that the simplest approach, *concatenation*, of placing one level segment after the other [196] can often fail, resulting in wasted computation. This leads to a generate-and-test approach where content is generated and tested to see if it matches the required specifications [186]. If it does not, new content is generated. This process repeats until valid content

[1] Code available on GitHub: https://github.com/bi3mer/LinkingLevelSegments
[2] Data generated for this Chapter available on GitHub: https://github.com/bi3mer/GramElitesData

is found. While not inherently problematic, this approach can be wasteful and slow.

In the case of this dissertation, the aim is to use the level segments generated by Gram-Elites and link them together to form a graph. Losing level segments because they cannot be linked via concatenation is simply unacceptable. Therefore, finding a strong answer to **RQ1.2** is important so that the segments generated by Gram-Elites can be used to form larger levels, which are then played by a player.

Looking at concatenation in more depth, a few problems are worth considering when building a replacement. Concatenation has no notion of in-game structures (e.g., doors in *Icarus* or pipes in *Mario*). One potential fix is to require padding on either side of a generated segment [69]; this implicitly handles the problem of broken in-game structures. However, this is not guaranteed to address completability. Further, it creates a repeating pattern that experienced players may notice.

The solution presented is *linking* for 2D tile-based games, which creates a *linker*—a tiny level segment that connects two level segments. The full level is the concatenation of the first segment, the linker, and the second segment. The full level should be *completable* (i.e., a player can get from the beginning to the end) and *unbroken* (i.e., a level should contain no broken in-game structures). A linker is *usable* if it results in a completable and unbroken level.

To generate usable linkers, linking uses two Markov chains and breadth-first search (BFS). This method relies on breaking a set of input levels into vertical or horizontal level slices. These level slices are used as input for two Markov chains. Both chains are used for structure completion for their respective level segment to guarantee an unbroken level by adding level slices to the linker—a limitation is that Markov chains can fail for unseen input. If we concatenate with the linker generated by the Markov chains, we can only guarantee that a generated linker will result in an unbroken level. To address completability, we use a tree search, which runs on the input level slices or a set defined by the designer. An agent is used at every iteration of the search to test if the output results in a level that is completable. The output of the search is an ordered list of level slices—the list may be empty. These are added in between any level slices added by the two Markov chains to form the linker. If a linker can be produced, it is guaranteed to be usable.

Linking and concatenation were tested with three games (*Mario*, *Icarus*, and *DungeonGrams*) with level segments generated by Gram-Elites for each game. Linkers always resulted in unbroken levels. Further, linking always found usable linkers between two segments for *Mario* and *DungeonGrams*. Concatenation worked well for *Mario*, but often failed for *DungeonGrams*—structures were broken, the level was not completable, or both. For *Kid Icarus*, linking almost always found usable linkers; concatenation was likely to fail. Testing linkers was extended to ana-

lyze how they performed when linking multiple-level segments. For *DungeonGrams*, we found that increasing the number of segments resulted in levels that were not completable due to the stamina mechanic. For *Mario* and *Icarus*, though, increasing the number of segments did not affect usability.

4.1 Related Work

This section is split into four parts. The first part discusses previous work that has combined vertical level slices and tree search to generate levels. The second part explains work that has connected dungeon rooms to form whole dungeons. The third part shows methods of level repair. The fourth and final part looks at work that used level segments to form larger levels.

4.1.1 Markov chains and Tree Search

2D tile-based game levels can be broken down from a grid to an ordered set of horizontal or vertical level slices. Dahlskog et al. [43] use these slices as input to an n-gram [91] to generate levels. Summerville et al. expanded on this by using slices as input for a Markov chain where the output can be more than one level slice [174]. To generate complete levels, they used Monte-Carlo tree search (MCTS) [39]. They added a twist where the MCTS was more likely to follow the probabilities of the Markov chain. This provides a better guarantee of the kind of levels produced. Tree search allowed Summerville et al. to produce levels that were completable and unbroken. Further, they guided the search to match target BCs.

4.1.2 Connecting Dungeon Rooms

Liapis [109] used evolution to generate a higher-order representation of a dungeon based on rooms. A secondary evolutionary process was run to build these rooms where every room had constraints (e.g., must have a connection to the left and right). These constraints were how rooms, or level segments, correctly linked to form a larger dungeon.

An alternative approach comes from Ashlock and McGuinness [9]. They used evolution to fill in a grid with different tile types. Dynamic programming was built into the fitness function to guarantee that a path existed between all level checkpoints, ensuring that the larger level was completable. They then used a room membership algorithm to find the evolved rooms and a separate algorithm to find adjacent rooms. Lastly, they filled the rooms with content based on required and optional content.

4.1.3 Level Repair

Cooper and Sarkar proposed one approach to level repair using a A* pathfinding agent [37]. The agent could do impossible actions, such as jumping through a solid tile. However, these actions had an extremely high cost associated with them. Therefore, the agent only used those actions if there was no valid path. Points in the path that were impossible to reach were repaired. For example, a solid block that the agent jumped through would be replaced with an empty block.

Another approach comes from Zhang et al. [213]. They used a generative adversarial network (GAN) [64] to generate a level but found that it did not reliably encode playability. To address this, they used a "generate-then-repair" framework. Levels were generated with the GAN and then repaired with mixed integer linear programming, which found the minimum number of changes that resulted in a fully playable level.

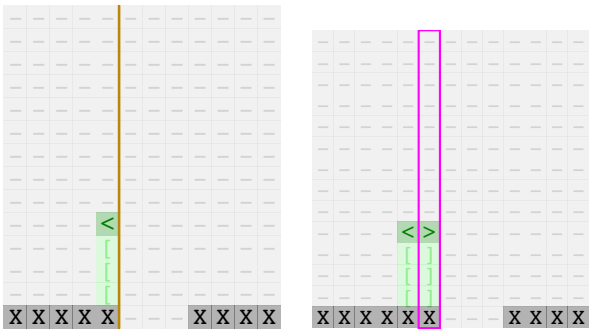
Jain et al. [87] used an autoencoder trained on *Mario* levels from the VGLC [176]. Among other uses, the autoencoder network repaired levels. In their work, a level was broken into windows or small level segments, which were input into the network. By placing a broken window into the network, they showed that the network would output a similar but playable segment.

4.1.4 Using Level Segments

Volz et al. [196] trained a GAN on a single *Mario* level. Their innovation in the process was to search the latent space of the generative network with covariance matrix adaptation evolution strategy (CMA-ES), an evolutionary QD algorithm that is often used as a comparison to MAP-Elites [75]. In one of their experiments, they tested forming larger levels with increasing difficulty. This was accomplished by generating the level segments with target difficulties and then concatenating them together, which means there was no guarantee that the resulting level was unbroken and completable.

Sarkar and Cooper [151] used variational autoencoders (VAE) [103] trained on levels with path information to generate level segments. The VAE was trained to encode a level segment and then used the encoding to predict the segment that followed. A level was generated by starting with one segment and then sequentially building new segments with the VAE; the VAE generated segments sequentially so that they connected.

Green et al. [69] started with a corpus of *Mario* levels that were generated with a required padding of two vertical slices of ground tiles on either side [100]. The segments were connected with the padding to form larger levels. Larger level sequences were formed using Feasible–Infeasible



(a) (b)

Figure 4.1: (a) The tan line marks where the starting segment ends and the end segment begins. There is an incomplete pipe with concatenation. (b) The magenta box shows the output of the forward chain which completes the pipe.

Two-Population (FI-2Pop) [102], an evolutionary algorithm for evolving two populations where one is infeasible (i.e., a function says that the solution meets does not some set of requirements) and the other is feasible (i.e., the solution meets the bare minimum requirements and is evaluated on a second function). In Green et al.'s work, infeasible sequences were optimized to be completable. Feasible sequences were optimized to match a target mechanics sequence. The padding guaranteed unbroken levels, and FI-2Pop found combinations of segments that were completable.

Li et al. [108] used an ensemble of Markov chains to generate *Mega Man* levels. They used a first-order Markov chain to model the direction (i.e., horizontal or vertical) of a set of rooms. They used two L-shape Markov chains [163] to generate the rooms, one for horizontal rooms and the other for vertical rooms. They placed these rooms together and ran a check to guarantee that a path from one to the other existed. If a valid path did not exist, they re-sampled until two rooms with a connection were found, guaranteeing a completable level.

4.2 Generating Linkers

When connecting two level segments, there are three problems:

1. Does the start segment end with an incomplete structure?[3]

2. Does the end segment start with an incomplete structure?

3. Does the linker result in a completable level when used to link the start and the end segment?

```
procedure BUILD_LINK(start, end)
    for s in forward_chain.get(start) do
        for e in back_chain.get(end) do
            m ← tree_search(start + s, e + end)
            if m ≠ None then
                return s + m + e
    return None
```

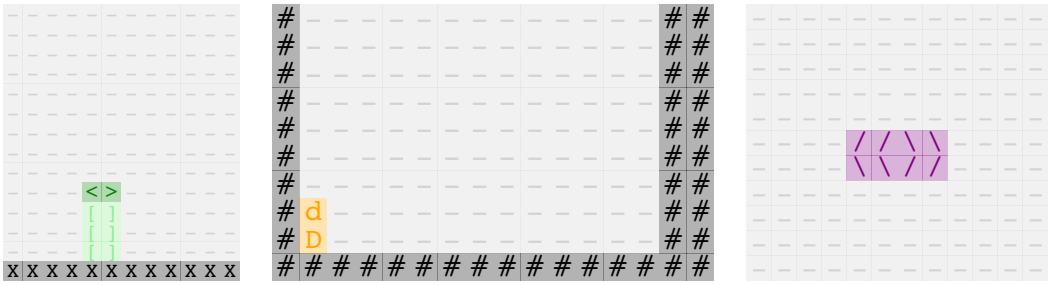
Figure 4.2: A pseudocode description for building a linker.

Let us start by addressing the first two problems. The simplest solution is to filter out level segments that start or end with incomplete structures. Filtering, though, is unnecessary and wasteful. The solution introduced in this Chapter is *structure completion*.

In some games, like *Mario*, checking for incomplete structures at the edge of segments and adding a custom level slice to complete them would be simple. However, there is a more general solution since we are guaranteed that the level segments being linked are n-gram generable: two Markov chains. The first, the forward chain, addresses potentially incomplete structures at the end of the starting segment. This chain is initialized from data in the direction the designer meant (e.g., *Mario* is from left to right). The second, the back chain, reads in reverse. The data that each Markov chain receives only includes level slices that contain tiles associated with in-game structures. This results in input and output that can be more than one level slice. Further, input is limited by the known size of a structure—e.g., *DungeonGrams* structures are four columns long—to prevent the potential problem of adding additional structures when only one is necessary.

Figure 4.1 shows the forward chain in action. It uses the last slice of the starting segment as input into the chain. If there is no output, then there are no incomplete structures at the end of the starting segment. If there is output, check for output for the last two slices. If so, check for three, and so on, until there is no output. The output associated with the most slices in the Markov chainis used. The back chain follows the same process but in the opposite direction. Once both chains run, it is guaranteed that the concatenation of the starting segment, forward link, back link, and end segment has no incomplete structures if the input segments are generable by the n-gram used in Gram-Elites, which comes from the data used to create the forward and backward Markov chains. If either chain receives a structure not present in the input data, structure completion with

[3]Figure 4.1a is an example of this problem.



(a) *Mario* (b) *Icarus* (c) *DungeonGrams*

Figure 4.3: Examples of structures for all three test games. *Mario* has pipes which are two columns wide and some number of rows tall. *Icarus* has doors that are one column wide and two rows tall. *DungeonGrams* has a structure that is 4 columns wide and 2 rows tall.

Markov chains will fail.

The next step is to address the problem of guaranteeing that a linked level is completable. *Linking slices* are level slices that do not contain in-game structures. By default, these are the level slices not input into the previously described Markov chains. These slices are placed into a unigram, which is effectively a list that can be iterated through. Alternatively, the designer can define them. BFS uses linking slices to find the minimum number of slices required to make the linked level completable. A benefit of using designer-defined linking slices is that the search space is reduced, and the algorithm is much faster. The output of BFS can be zero slices, but we can add constraints (e.g., require at least k slices). We use a max-depth check to prevent an infinite search.

View Figure 4.2 for the pseudocode of the proposed method. The `for` loops for both chains allow for the possibility that there may be multiple ways to complete a structure.

4.3 Method

This Chapter compares the proposed linking approach to concatenation for the three games tested in Chapter 3 (*DungeonGrams*, *Icarus*, and *Mario*) with the level segments that were generated by Gram-Elites. Figure 4.3 shows the structures in all three games. For all three games, a digraph was built using neighboring cells from Gram-Elites, based on the four cardinal directions, in the MAP-Elites grid using both concatenation and linking, and the results were compared.

For each game, the training levels used for n-grams in Gram-Elites were used as input for the forward and backward Markov chain used for linker generation. A filter was run to only include level slices with a tile associated with an in-game structure.

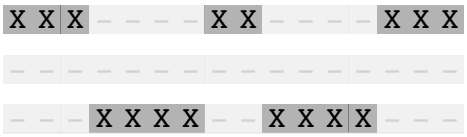


Figure 4.4: The three linking slices used for for *Kid Icarus*.

Each game was configured to use different linking slices—level slices used by the completability tree search—to demonstrate that an automatic approach was possible, but that an intervention would reduce the search space and yield faster results.

4.3.1 *Mario*

The linking completability tree search used default linking slices with no designer intervention. As a reminder, the default linking slices did not include level slices with tiles related to in-game structures. The max depth of the linking tree search was 7. For the agent, a modified Summerville A* agent [176] was used to calculate the furthest point in the level that can be reached. Pipes are the only in-game structures; see Figure 4.3a.

4.3.2 *Icarus*

The linking completability tree search used one empty row and two rows with platforms designed to connect to anything else; see Figure 4.4. The max depth for the linking tree search was 7. The agent for the tree search was the modified Summerville A* agent [176] to calculate the furthest point in the level that can be reached. *Icarus* has one structure which is a door, see Figure 4.3b.

4.3.3 *DungeonGrams*

The linking completability tree search used three columns with food; see Figure 4.5. The max depth of the linking tree search was 4, which was lower than the other two games due to the agent being slower due to a larger search space. *DungeonGrams* also has one structure that can be seen in Figure 4.3c.

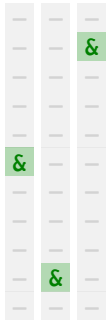


Figure 4.5: The three linking slices used for for *DungeonGrams*.

4.4 Evaluation

The evaluation of linking is broken into two parts. In the first part, the test case of linking two level segments for *DungeonGrams*, *Icarus*, and *Mario* is shown. In the second part, the linkers made to link two level segments are used to create even larger levels with three, four, and five level segments.

4.4.1 Concatenation Versus Linking

Segments generated by Gram-Elites are organized by the MAP-Elites grid where each axis represents a BC [162]. Rather than selecting random segments and linking them together, the grid is used to select similar segments by using neighboring segments—there are four possible neighbors in the four cardinal directions. There are $13,182$ possible links for *Mario*, $9,453$ for *Icarus*, and $6,086$ for *DungeonGrams*.

Table 4.1 shows the percentage of concatenations and links found that were unbroken, completable, and usable (i.e., both unbroken and completable). With *Mario*, concatenation was 92% likely to result in a level that was completable by an agent and 89% likely to result in a usable level, showing that concatenation is likely to be an effective strategy for most use cases in a simple platformer. Linking was successful for every possible link. Figure 4.6 shows two examples where concatenation failed. The top example shows that concatenation can fail when a gap is too large. The bottom example shows the largest linker required for *Mario*. The linker completed two structures, but did not affect completability, as the concatenated version was also completable.

Concatenation performed well for *Icarus* in producing an unbroken level. However, it was the worst at generating a completable level at just 11%. Linking, though, found a usable link

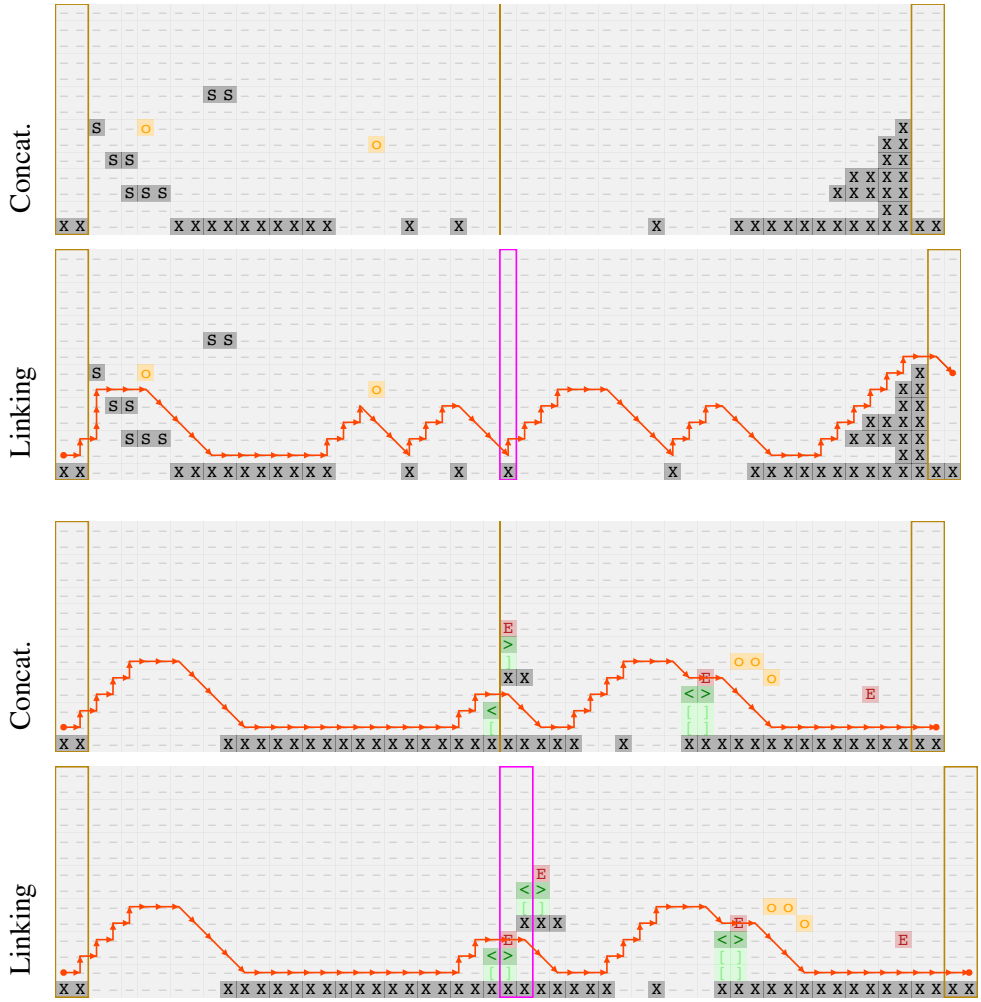
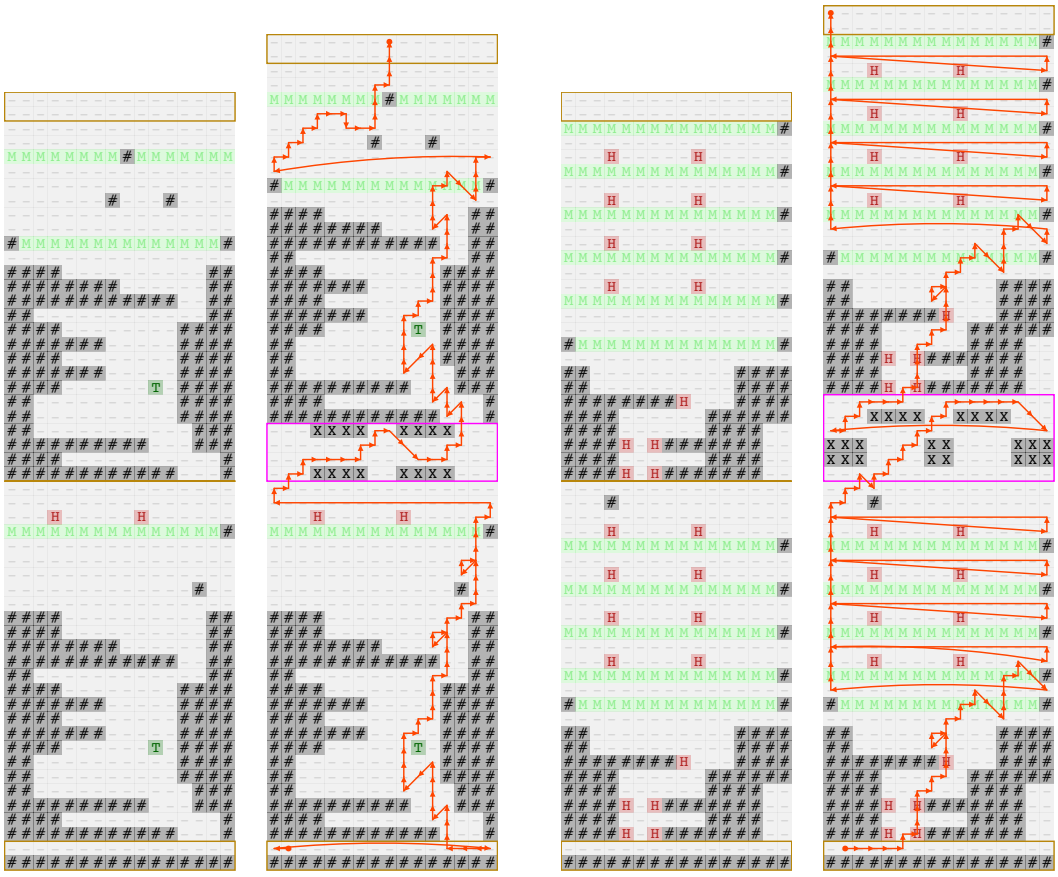


Figure 4.6: Two examples for *Mario*. The example on top shows concatenation failing due to a large gap. The bottom displays the longest linker produced. The tan line in the middle is where the link would have been placed for concatenation. The tan boxes on the left and right show the padding. The magenta box in the middle shows the linker found. The red lines show a path through the level if it was beatable.



Concat. Linking Concat. Linking

Figure 4.7: Two examples for *Icarus*. Both show concatenation failing due there being no possible path to complete the level. The right example of concatenation and linking shows the longest linker required to make a level completable. Padding is shown with the tan boxes at the top and bottom of the level. The tan line in the middle is where the link would have been placed for concatenation. The magenta box shows the linker found. The red lines show a path through the level if it was beatable.

	Type	Unbroken	Completable	Usable
Mario	Concatenation	0.97	0.92	0.89
	Linking	1.00	1.00	1.00
Icarus	Concatenation	0.96	0.11	0.11
	Linking	1.00	0.99	0.99
DG	Concatenation	0.52	0.58	0.23
	Linking	1.00	1.00	1.00

Table 4.1: Percentage of links that resulted in unbroken levels, completeable levels, and usable levels. DG is short for *DungeonGrams*.

for almost every single starting and ending segment given. Figure 4.7 shows two examples where concatenation could not produce a completable level. In both cases, the agent did not have enough space to make the required jump. Tree search resolves this by adding two platforms for the agent. The right-most example shows the largest linker built.

Concatenation performed poorly for *DungeonGrams* and was unlikely to produce a usable level. This was due to two reasons. First, the size and complexity of the structures made it unlikely for two segments to line up if either or both level segments had unfinished structures. Second, the concatenated level was impossible to beat if the two segments did not have enough food. Linking, though, handled both problems and always produced a usable link. Figure 4.8 shows two examples where concatenation failed. In both cases, structures were incomplete, and there was not enough food for the agent. Linking succeeded by completing the structures with the forward and backward chains, and tree search added food in the middle to make the level completable. The bottom example shows the largest linker found for *DungeonGrams*.

The usability of a linker is not the only important metric to consider. Ideally, a linker should not be identifiable or distracting to the player (e.g., the same set of level slices between every segment). To examine this, we report on the lengths of linkers found and the change in BCs of a two segment level when a linker is used. We determine the latter by finding the BCs of the concatenated

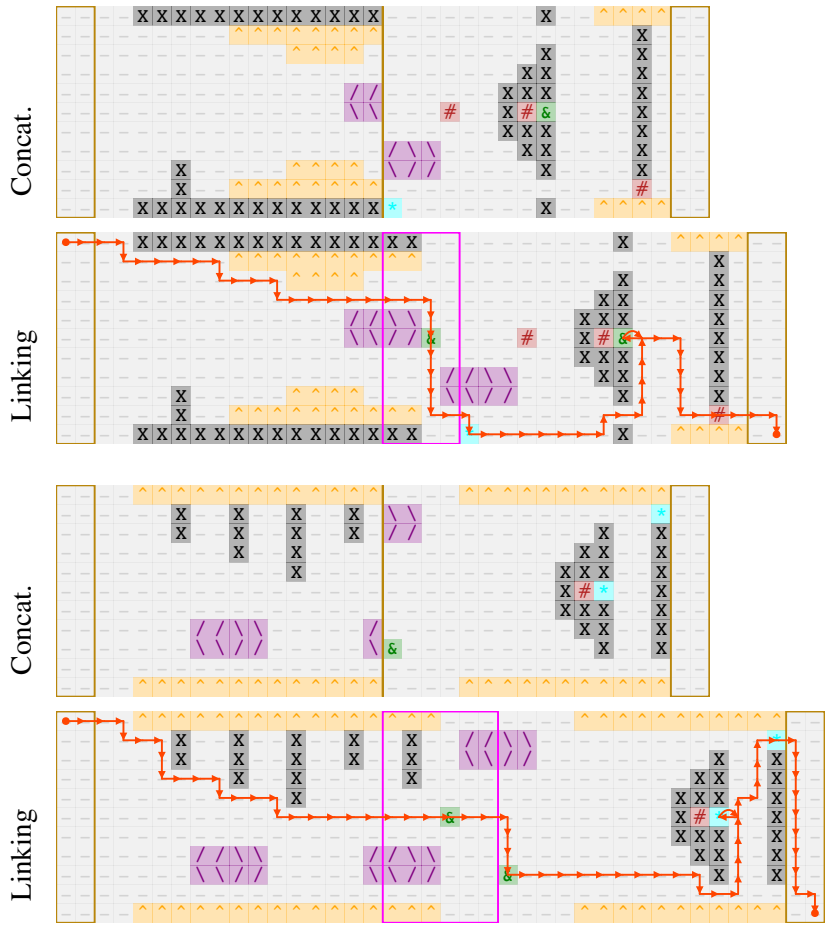


Figure 4.8: Two examples for *DungeonGrams* where concatenation fails due to the agent running out of stamina. (& represents food.) The bottom concatenation and linking example shows the largest linker found. The tan boxes on the left and right show the padding. The tan line in the middle is where the link would have been placed for concatenation. The magenta box shows the linker found. The red lines show a path through the level if it was beatable

Game	Mean Length	Median Length	Max Length	Mean D_{BC}	Median D_{BC}	Max D_{BC}
Mario	0.110	0	2	0.001	0.000	0.020
Icarus	2.658	3	6	0.008	0.008	0.029
DG	1.147	1	6	0.012	0.011	0.061

Table 4.2: For each game, shows the mean, median, and max for lengths of linkers found and D_{BC}. DG is short for *DungeonGrams*.

Segments	*Mario*	*Icarus*	*DG*	*DG-Food*
2	1.00	1.00	1.00	1.00
3	1.00	1.00	0.99	0.99
4	1.00	1.00	0.98	0.99
5	1.00	1.00	0.97	0.99

Table 4.3: Shows the percentage that linked levels are usable. DG is short for *DungeonGrams* where an empty linker is allowed. *DG-Food* requires that the tree search adds at least one level slice to the linker.

level and the linked level and calculate the Euclidean distance between the two. The result is D_{BC}.

The results can be seen in Table 4.2. For *Mario*, we can see that a linker tended not to be required, and when it was, the worst case was a linker with two-level slices. As a result, D_{BC} must be small. To put the value in context, the implementation of Gram-Elites tessellated the grid by changes in BC every 0.025 for *Mario*. For *Mario*, if a linked level and its corresponding concatenated level had a D_{BC} smaller than 0.025, then both were similar.

Icarus was different because it almost always required a linker with two to three linking slices. The max length was also much longer than *Mario*. The increment for *Icarus's* grid was 0.0125. The mean and median of D_{BC} showed that most linkers were unlikely to cause a large changes to the BCs. The maximum D_{BC} shows that the worst case was a jump of two to three neighbors in the grid.

Finally, *DungeonGrams* on average had smaller linkers than *Icarus* but had the same max linker length. The mean and median of D_{BC} was relatively small: the Gram-elites grid was 0.05 per bin in *DungeonGrams*. So, the worst case was a change of one cell in the MAP-Elites grid.

Overall, linking usually resulted in minimal modifications to the player's experience per segment while providing the benefit of completability and unbrokenness.

4.4.2 Linking Multiple Segments

In this section, linkers made for two segments are tested by combining multiple segments. One thousand random levels were generated by following the MAP-Elites grid, neighbor by neighbor. Linkers were selected if they had a playability guarantee for the starting and ending level segments. The results are in Table 4.3.

Increasing the number of segments for both *Icarus* and *Mario* did not change the results:

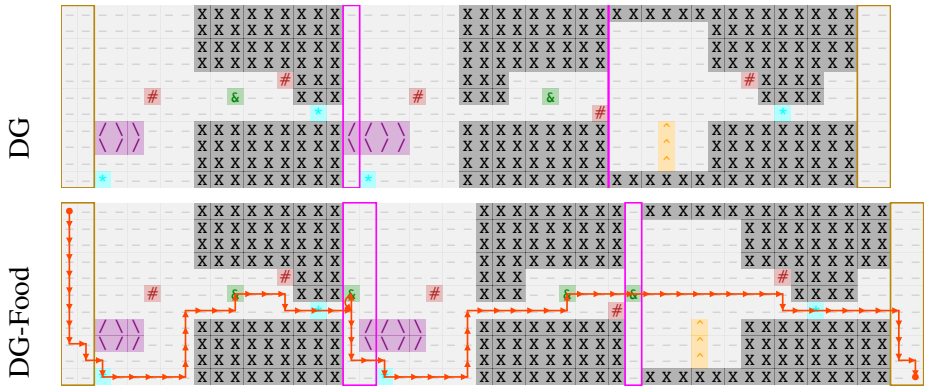


Figure 4.9: Example of three segments being linked in *DungeonGrams*. Tan boxes on either side represent padding. Magenta boxes in the middle represent a linker, and the magenta line shows that no linker was necessary for the second and third level segment for linking (*DG*). The agent fails to reach the end for *DG* due to a lack of food, but this was not a problem for linking with food required (*DG-Food*).

all the linked multi-segment levels were usable. This was an unsurprising result because neither game has long-term dependencies. In this case, a long-term dependency refers to a resource that the player must use to progress through the game. While neither platformer has this, *DungeonGrams* does: stamina.

For *DungeonGrams*, two linking approaches were tested:

- *DG* - an empty link was allowed.
- *DG-Food* - at least one linking slice was required as output from the tree search.

Recall from Figure 4.5 that each column had at least one food tile. *DG-Food* requires one linking slice, meaning that *DG-Food* will always have at least one food tile between the level segments it links.

DG performed increasingly poorly compared to the platformers as the number of segments increased. In contrast, the simple change for *DG-Food* results in larger levels almost always being fully playable. Figure 4.9 shows an example where *DG* fails and *DG-Food* succeeded. Failure occurs since linking guarantees that a path exists between two segments but does not ensure that the player will have the same stamina at the start of a new segment as they do when starting the game. While food in *DungeonGrams* does not give the player back their max stamina, it does increase the likelihood that the agent can make it through the next segment.

4.5 Limitations

- Linking relies on forward and backward Markov chains. If given a start level segment that ends with a structure not in the forward chain's input data, linking will fail. The same problem exists for the end segment if the start of the end segment includes an incomplete structure.

- If linking were to be used in an online setting, it has one major limitation: the agent. For linking to be useful, it needs to provide a completability guarantee. If the agent is slow to test whether a linked level is completable, then linking is slow. An approximation of whether a level is completable could be accomplished with neural networks or heuristics [10], but then the developer runs the risk of a misclassification and serving levels that cannot be beaten, which would surely frustrate the player.

- Whether using designer slices or not, if BFS adds a level slice to the link, there is no longer any guarantee that the full level will be n-gram generable. Given the small size of linkers and the shown minimal changes to BCs, this is not a big problem, but it is a limitation that linkers do not guarantee n-gram generability.

4.6 Unexplored Areas of Future Work

- Linking, as described, works perfectly for games that do not have long-term dependencies. However, it fails for games like *DungeonGrams* if the links built are to be used for creating levels of more than two level segments. An approach not explored in this Chapter is building a linking algorithm that can handle n input level segments. The process is the same, where structure completion goes first and then a BFS runs until the full level is completable, except it would be done for n level segments. The system would have to note where the agent died in the playthrough and use that information to direct the search.

- The linking approach does not modify the level segments but adds additional level slices. However, an alternative approach is to see the problem as a level repair problem. Instead, 2D Markov chains could be used to repair structures, and a pathfinding agent could be used to add blocks until the level was completable [37].

- The linking approach works by first adding level slices associated with structures and then adding level slices associated with completability. This second step makes the levels longer.

In *DungeonGrams*, this means that food is further away, and so is the exit, thus making it more difficult to complete the level. Instead, it is possible that the slices used for structure completion could have blank tiles overwritten with food. This could be accomplished by testing if the linking slice has non-blank tiles compared to the structure slice, which has blank tiles. If so, an overwrite is possible and can be tested. This could have resulted in smaller levels and better overall performance.

4.7 Conclusion

This Chapter answers **RQ1.2**: How can usable level segments be linked together to form levels? The problem was examined and broken down into two requirements: fixing broken structures and making the full level completable. The first problem was solved with two Markov chains, a forward chain and a back chain, and the second problem was solved with BFS on level slices with a completability agent.

The linking approach was tested for three games—*DungeonGrams*, *Icarus*, and *Mario*—and compared to concatenation. Linking outperformed concatenation for all three games. Only for *Icarus* did it not find a link 100% of the time, and for *Icarus*, it was at 99% whereas concatenation was at 11%.

The links built were then tested for linking multiple segments. The links worked perfectly for *Icarus* and *Mario* but performed worse for *DungeonGrams*, which has a long-term dependency mechanic, stamina. An alternative linking approach was tested where one column with food was required in between level segments, and the results improved but did not reach 100% success.

We now have level segments and links between them, forming a complete digraph. The next step is to see how this digraph can be used to assemble levels for DDA.

Chapter 5

Level Assembly as a Markov Decision Process

> Imagining something is better than remembering something.
>
> John Irving, *The World According to Garp*

This Chapter is based on the paper, "Level Assembly as a Markov Decision Process" [18].[1] The authors are myself and Seth Cooper.

The overall research question for this Part of the dissertation is **RQ1**: How can usable levels be assembled from level segments for dynamic difficulty adjustment? Recall that this question was broken into two sub-questions.

- **RQ1.1**: How can usable level segments be generated within an organized structure applicable to dynamic difficulty adjustment?
- **RQ1.2**: How can usable level segments be linked together to form levels?

Chapter 3 answered the first subquestion. Gram-Elites was introduced as a method to generate level segments that were *usable*. Chapter 4 answered the second subquestion. A solution that utilized two Markov chains for structure completion and tree search with a playability agent for completability was introduced to build links between level segments. The combination of level segment, link, and level segment allowed for creating larger levels that were *usable*. This leaves the overall question of **RQ1** left to answer.

[1] Code available on GitHub: https://github.com/crowdgames/mdp-level-assembly

The end result of answering **RQ1** will, in effect, be a replacement for a SLP, and it is worth considering why games use SLPs. At face value, a dynamic approach appears preferable because it can tailor the game's experience to each player. So, why are SLPs the standard approach? First, consider the workload of creating a SLP. A static progression requires a minimal number of levels. DDA via PLG requires multiple levels for different players, requiring more work from already overworked developers [44]. A potential solution is procedural content generation [153], but designers will still need to do additional work [94]. The second problem is adapting to the player. A player model [81, 208] could 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. One last scenario to consider is that the designer does not want the game to adapt to the player. An example of this is a souls-like game, such as *Elden Ring* [55], where the difficulty and overcoming that difficulty is a major part of the game's experience. Alternatives to SLPs are either (1) too much work or (2) not relevant to the game. This dissertation and Chapter aim to show that there is a method for DDA via PLG that does not require too much extra work from developers.

The solution presented for **RQ1**, is to formulate the digraph generated in Chapter 4 as an MDP—see Section 2.6 for an introduction to MDPs. The rewards of the MDP are built using a custom reward function built with BCs to represent the reward.[2] Reward, in this case, can be seen as the desirability, stated quantitatively, of the designer that the player plays a given level segment. The reward table updates based on a dynamic reward function that considers designer and player preference as well as how many times the player has played a given state. The *director* selects the next state—level segment—based on the current state. Levels are formed by concatenating the director's selection of level segments, using links when required. Before assembling a level, the MDP is updated based on the player's performance in a previous level. Therefore, the director can adapt to the player. Four directors—described in Section 5.2.2—were tested:

- Random
- Greedy
- Policy Iteration (PI)
- Adaptive Policy Iteration (API)

[2] BCs are not necessarily the best approximation of rewards if this system were to be used by players instead of agents. Chapter 7 is devoted to finding better rewards.

Random and Greedy are both baseline directors. PI uses an unmodified version of policy iteration—see Section 2.6.2. API—one of the contributions of this Chapter that is described in Section 5.2.2.4—removes edges from the MDP—i.e., make an action invalid at state X—if the player is performing poorly. After this step, API runs policy iteration just like PI.

To test these directors, we ran two case studies. The first case study questions the assumption that generating levels with pre-generated level segments is the best approach. The MDP in this case study is initialized with the n-gram used in Chapter 3 for *Mario*. Rewards are based on whether an enemy was present in a given output level slice. An A* agent tested completability, and a surrogate player model rewarded level slice density. API and PI performed equally well regarding reward, but API produced levels that were, on average, more completable. Random was the best performer in terms of completable levels but worst for reward. Overall, the results were interesting, but repeating patterns of level slices made level generation via an n-gram appear undesirable for a game to be played by actual people. Further, an agent could not complete some levels generated, which is unacceptable for a commercial PCG system.

The second case study used an MDP that connected *Kid Icarus* level segments to form a level. We used player proxies [68, 110] to evaluate level segments, where a proxy has a distribution over the levels it could beat and where it struggled. A proxy also specified the kind of level segments it enjoyed as a player reward. We found that API performed best on average across all the player proxies. Additionally, we tested how well directors adapted when a player proxy changed. We found that only API adapted to the change.

5.1 Related Work

5.1.1 Adaptive Games

One of the first cases for adaptive games comes from Hunicke [84], which argued for DDA. Hunicke showed how small adjustments and interventions improve player performance. DDA can also be achieved through level generation, and Jennings-Teats et al. [90] gave an early example with *Polymorph*. They used a generator [161] 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 and the player's subjective difficulty ranking were stored. They used this data to train two models to classify difficulty and player skill. By adopting a generate-and-test approach, the game adapted to the player.

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, which leads to the conclusion that online learning is a more promising approach for dynamic progressions because it addresses the weaknesses of pre-built models. However, there are problems with how quickly these algorithms can adapt to the player.

Linden et al. [192] followed Polymorph's footsteps by building a graph grammar around potential player actions to generate dungeons. For example, the action "open door" can be defined by sub-actions such as "kill enemy" followed by "loot key from body." The output of the grammar was fed into a level generator that built a level based on the required actions to beat the level. The grammar can also be modified with global conditions, such as difficulty level. For example, the grammar may have two possible paths to get a key, but one where the player has to slay a dragon will only be mandatory if the difficulty is over a certain threshold. This threshold can be modified based on player performance to make the game easier or harder.

Another approach to DDA is to optimize the progression to maximize a player's number of levels played [206]. Xue et al. built a probabilistic graph based on nodes representing a level and how many trials a player took to beat said level. This graph is solved with dynamic programming. They used this system to deliver levels to users in real-time in a newly published game and found that their system, instead of randomly selecting levels, resulted in more rounds played and more time spent in the game.

One approach to address the limitations of online learning techniques comes from Gonzalez-Duque et al. [62]. In their work, they used MAP-Elites [129] to generate a grid of levels for multiple agents, where the objective was 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 [41]. In the ideal case, their system adapted to a new agent in one iteration. Overall, this is similar to the work to the work in this Part of the dissertation. 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.

5.1.2 Markov Decision Processes for Games

One of the earliest formulations of a game as an MDP comes from Thue and Bulitko [183] with the procedural game adaption (PGA) framework. The MDP decided what happened next in

the world for a player. Outside of the MDP, the *manager* estimated the player's reward function and policy. With these two models, the manager updated the MDP. Thus, player decisions affected the MDP, which was optimized in real-time to give the player the best experience. Thue and Bulitko did not directly apply PGA to DDA. Instead, they focused on narrative direction and player preferences (e.g., weapon preference).

Massoudi et al. [118] used an MDP with SARSA [144]—an on-policy reinforcement learning method—to choose the best actions for an enemy based on the desired difficulty for a player. Mandel et al. [116] used a Partially Observable Markov Decision Process built from player data that was solved with an offline policy evaluation technique and deployed to an educational game to optimize player engagement.

Shu et al. [159] built a framework called experience-driven procedural content generation via reinforcement learning, which is composed of four systems: (1) a game-playing agent, (2) a latent space generator (e.g., a GAN), (3) a level repairer, and (4) experience model. This framework was designed based on the structure of a MDP. A state was a level. An action was a latent vector input to a GAN to get a level. A neural network modeled the transition model and output a new state given the previous state as input. A player experience model defined the reward.

Another approach to level generation with a MDP comes from Sheffield and Shah [156]. In their work, they formulated level generation as a MDP. The MDP was built from observations gathered while a designer built a level. Those observations were used to build a policy with apprenticeship learning via inverse reinforcement learning [1].

5.1.3 Intelligent Tutoring Systems

An Intelligent Tutoring System (ITS) is an AI system for computer-based instruction [38, 65, 135].[3] The goal is player/student learning. The same can be said for a DDA system, because the goal is for the player to overcome increasingly difficult challenges. The main difference is the kind of content: an ITS may want the player to learn multiplication [157] or grammar [6], while DDA can be applied to any genre of game, including educational games. A goal for both is also to make sure that the game does not become so difficult that the challenge becomes too much or not enough such that the player quits, as shown in Section 5.4.3.2.

McNamara et al. introduced Intelligent Tutors and Games (ITaG) [122]. In their work, they looked at how ITS can incorporate elements of games. They started by breaking down the term

[3] AI is used in its broader definition here, where the system does not need to be a neural network trained on the entirety of the internet [22].

"motivation" into four components:

1. Self-regulation: The student actively seeks a coherent understanding of the topic.
2. Self-efficacy: The student has a sense of success and achievement.
3. Interest: The degree to which the student's underlying needs and desires are energized.
4. Engagement: The degree to which the student is not bored. The opposite of boredom is flow [42] or high engagement.

These four components were used to examine elements of games that could be incorporated into an ITS. They specifically looked at feedback, incentives, task difficulty, control, and environment. McNamara et al. noted that incorporating game elements from these categories would not *directly* influence learning. However, they expected to observe the indirect benefits of an ITaG, such as higher levels of engagement for longer periods and increased motivation. The effect of these indirect benefits should be increased student learning. This is directly related to the work in this Chapter because the goal of the director is to maintain player motivation via continuous challenge until the player starts to fail. When the player begins to fail, API adapts by making the experience easier before returning to a more challenging experience for the player.

Mitchell et al. used a MDP to model when a tutor should intervene with a message [126]. Their problem space was computer science students working with a tutor on a programming task, where the tutor could see the student's screen and send messages back and forth with the student. They recorded these and made a state s the combination of three features: (1) the current student action, (2) task trajectory, and (3) the last action taken by the tutor. An action was either the tutor sent a message or did not. Before and after the session, the student was given a survey, and Mitchell et al. used these to measure normalized learning gains. They added three additional states to the constructed MDP: (1) initial state, (2) terminal state 1 (+100, the student had learning gains higher than the medial), and (3) terminal state 2 (-100, the student did not have learning gains higher than the median). They then ran policy iteration on the resulting MDP and analyzed the results. They found that the tutor should engage with the student when the student was engaged but should not send more than one message.

The method presented in this dissertation to answer **RQ1** does not directly build on the body of literature on intelligent tutoring systems. Instead, an assumption is made that the "simpler" levels, based on heuristics in Chapter 3, will use fewer mechanics. As the player moves to more

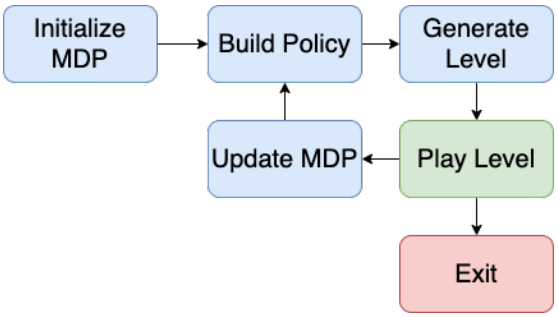


Figure 5.1: A flowchart of the overall level assembly system.

"complex" levels, again based on the same heuristics, the player will be naturally introduced to new mechanics. However, this relies on an assumption and not a guarantee. This will be partially addressed in Chapter 9.

5.2 Approach

5.2.1 Level Assembly via a Markov Decision Process

A diagram of the system is shown in Figure 5.1. The system starts by *initializing the MDP*. A Markov Decision Process (MDP)—see Section 2.6—is a framework for modeling decision-making for discrete-time and known-state environments.[4] The MDP, in this case, builds a level assembly policy that is tailored to the player. For this work, we added two states: $start$ and $death$. The death state has a reward of -1 and is the only state with no outgoing edges. $start$ is where level assembly begins.

Due to using a digraph as a starting point, the MDP is represented with a graph instead of a matrix [125], which is relevant for the transition table and actions. A state is equivalent to a node, and an action to an edge. Given this, the transition model is better represented as $P(s'|s, s_{tgt})$—s is the starting state, s_{tgt} is the target state given the edge, and s' is the state actually traversed to—where $P \in \mathbb{R}$ is in the range of $[0, 1]$ and is normalized. Each edge can result in the player beating and entering the target state or losing and entering the *death* state. The player's probability of successfully beating the target state is initialized 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.

[4] If all possible states are not known ahead of time, a partially observable Markov Decision Process is required [113, 166].

In the case studies below, each MDP is discussed in detail, but for now, assume the existence of an MDP. In addition to $R(s)$, there is 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; metrics that can be automated are used in this work, such as counting the number of enemies in a level slice. $R(s)$ is initialized with $R_D(s)$: $\forall s \in S : R(s) \leftarrow R_D(s)$. In this scheme, the MDP starts with a bias toward the designer and, as shown below, updates over time.

The second step is to *build the policy* for level assembly. A policy $\pi(s)$ chooses a connecting state given the current state as input. Section 5.2.2 describes four different directors and how they build a policy.

The third step 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.[5]

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, the MDP is updated based on the results of the previous play session.

The process starts by modifying the MDP. Any level segment the player completed is added as a neighbor to the $start$ node of the MDP. In this case, the player only beat s_1. If s_1 was not a neighbor of the start node, the edge $(start, s_1)$ would be added—probabilities are handled below. In this case, s_1 was already a neighbor of the start node because the policy selected s_1 from the start node. As a result, no edge was added.

$$R(s) \leftarrow \frac{R_D(s) + M(s)}{N(s)} \tag{5.1}$$

The next step is to update $R(s)$ for visited states—s_1 and s_2—with Equation 5.1. This equation assumes the existence of $M(s)$, a player model that quantifies player enjoyment of a given segment after gameplay.[6] $N(s)$ tracks the number of times a state was visited—all state visit counts are initialized to 1 to avoid a divide by zero error. 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 [67].

[5] If there is a linker between the two states, it would also be included.

[6] Note that this player model is not strictly necessary. In fact, it will not be used in Part III at all, which uses players instead of agents. The player model is included to show how this system can be extended based on the designer's wants and needs.

$$P(s_{tgt}|s_{src}, s_{tgt}) \leftarrow \frac{1 + \sum_{\{s,s_{tgt}\}\in E} \textit{wins}(s, s_{tgt})}{1 + \sum_{\{s,s_{tgt}\}\in E} \textit{visits}(s, s_{tgt})} \tag{5.2}$$

The last step is to evaluate transition probabilities. E is the set of all edges in the digraph. The number of times an edge has been used and the number of times the player has beaten the target state is updated. In the example, we updated for two edges: $(start, s_1)$ and (s_1, s_2). Visits are incremented for both edges, but only $(start, s_1)$ has the win count incremented. Then P is updated with Equation 5.2 for every edge with the target state of s_1 or s_2. Lastly, the probability of entering the death state for every affected source state in the digraph is updated: $P(death|s_{src}, s_{tgt}) \leftarrow 1 - P(s_{tgt}|s_{src}, s_{tgt})$.

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

5.2.2 Directors

A *director* [207] 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.

5.2.2.1 Random

Random, a baseline director, builds a policy by selecting a random neighbor for each state s.

5.2.2.2 Greedy

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

5.2.2.3 Policy Iteration (PI)

PI, or policy iteration, runs policy iteration—see Section 2.6.2—on the MDP to generate a policy π^*.[7] The parameters used were $\gamma = 0.95$ and $k = 20$, both found experimentally by running small trials. The resulting policy takes an input state and returns an output state.

[7] Convergence for an MDP with $9,453$ states takes < 0.5 seconds running unoptimized Python code.

5.2.2.4 Adaptive Policy Iteration (API)

PI struggled in case study 2 to adapt when a player's persona changed. 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 level segments that cause continuous player failure.

5.2.3 Evaluation

Two case studies evaluate the described approach. The first case study models n-gram level generation as an MDP. This task is challenging because n-grams may not have the full context required to make a level that an agent can complete. The second case study uses the digraph generated in the previous two Chapters and turns that digraph into a MDP. This guarantees completable levels, and the MDP can optimize towards the player. Additionally, the second case study has a larger MDP in terms of states, which helps stress test whether the system is viable for running in a real-time situation with players instead of agents.

5.3 Case Study 1: *Mario* N-Grams

5.3.1 Markov Decision Process

The n-gram used in Chapter 3 for *Mario* was used to form an MDP. Each state in the MDP represented a prior in the n-gram. An action was the output level slice of the n-gram, and these were represented with edges in the graph. There were 513 states with an average of 1.579 actions per state with a max of 47 actions for one state. The designer reward ($R_D(s)$) was 1 if there was an enemy in the state and 0 otherwise.

5.3.2 Players

Levels assembled with the described MDP were not guaranteed to be completable. The agent used to assess *Mario* levels in the previous chapters again assessed the percent that a level

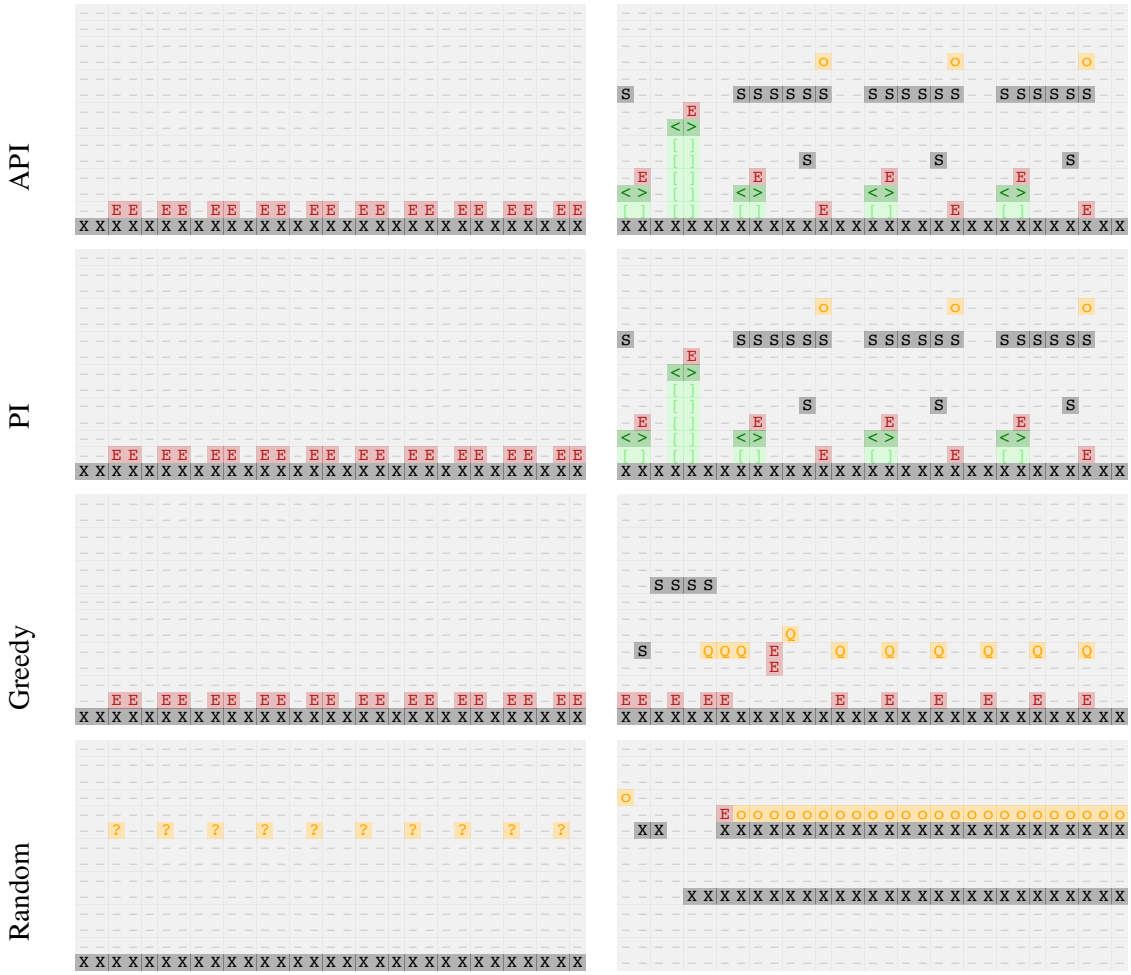


Figure 5.2: The first and last level generator by each director for *Mario*, where the player model liked levels with high density. Recall that the agent used was a perfect player, so player skill was not a limiting factor for this case study with *Mario*.

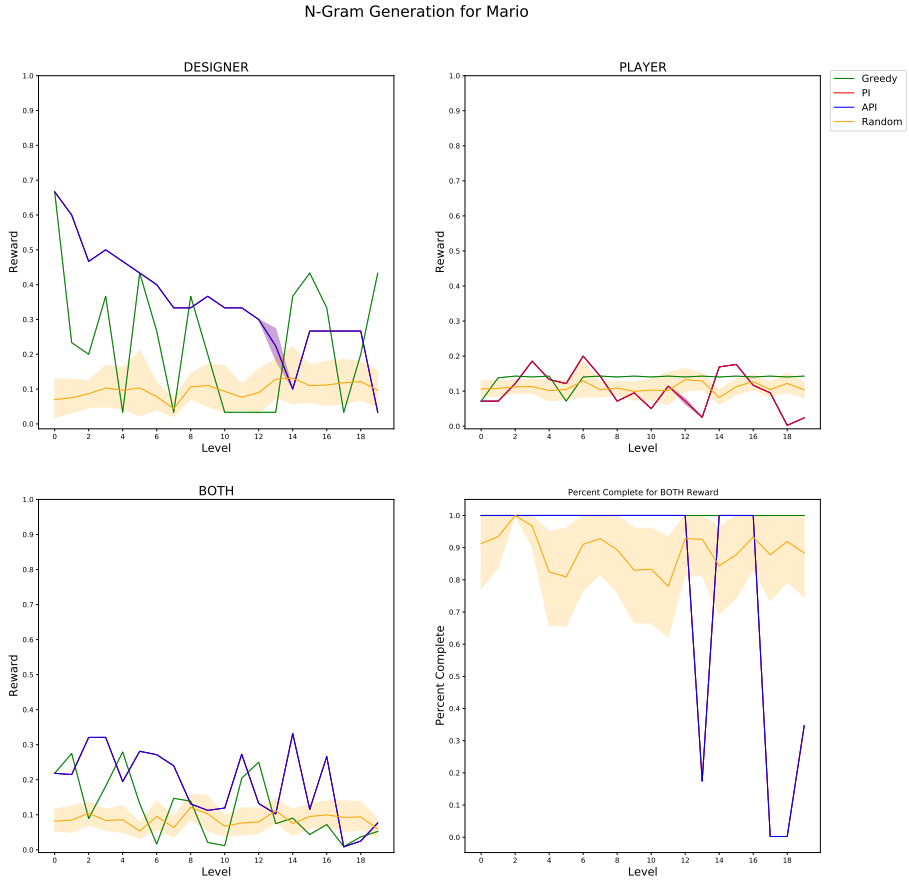


Figure 5.3: Average reward for all three reward types when running n-gram generation for *Mario*, and the average percent completable for the *both* reward..

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 5.1: 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.

could be completed. As a substitute for a player model ($M(s)$), player enjoyment was defined 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).

5.3.3 Evaluation

Each director ran twenty times on different seeds, generating fifty levels for each run. Fifty levels was chosen because it is 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 Figure 5.2. One problem seen for both API and PI was the prevalence of repeating patterns of level slices. Section 5.6 discusses an approach to address this. However, these patterns—and the fact that some levels generated were not completable—make level generation via a MDP unappealing to use with real players.

The overall results are in Table 5.1, and graphs are in Figure 5.3. In terms of reward, both API and PI were the best performers, with an identical average reward and standard deviation. Greedy was the third-best performer. Random was the worst. Interestingly, Greedy always produced a completable level. This resulted from getting stuck in a local maxima, which helps explain its low average reward. Random performed better in terms of completability than both API and PI. API produced levels that were, on average, more completable than PI. This was expected behavior because API removed edges to states that were problematic to start from.

5.4 Case Study 2: *Icarus* Segment Generation

5.4.1 Markov Decision Process

The MDP used for this case study was built from the digraph built in Chapter 4, which used the level segments built in Chapter 3 for *Icarus*. If a linker was required, a linking state was between the two states being connected. There were $9,453$ states and an average of 1.877 actions per state with a max of 11. However, there were many linking states, and when removed, there were $1,094$ states with a mean of 8.579 actions per state and a max of 11—this is possible due to their being multiple elites per cell in MAP-Elites.

The values of BCs for *Icarus* were between 0 and 1 in Chapter 3. For non-linking states, $R_D(s)$ was assigned the mean of the state's BCs. For linking states, $R_D(s)$ was the mean of the designer reward for the two linked states—this made it fairer for the Greedy director. Note the assumption here that the larger $R_D(s)$ 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 this is discussed more below.

There are three final points:

- Multiple elites per cell were very similar but had their own state. This was addressed by incrementing $N(s)$ for all states in that cell for reward decay.

- The work below has directors generate levels that are five level segments long. Therefore, a five segment level could be made from nine total states: five level segment states and four linking states.

- Linking states could not have an incoming edge from the $start$ state.

5.4.2 Players

The level segments combined are guaranteed to result in a level that an agent can complete. Instead of evaluating whether a level can be completed, player proxies [110, 68] were used to evaluate the MDP. A player proxy defined the level segments it could beat and how much it enjoyed that level segment. Table 5.2 shows the player proxies used.

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 5.2: 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, $M(s)$.

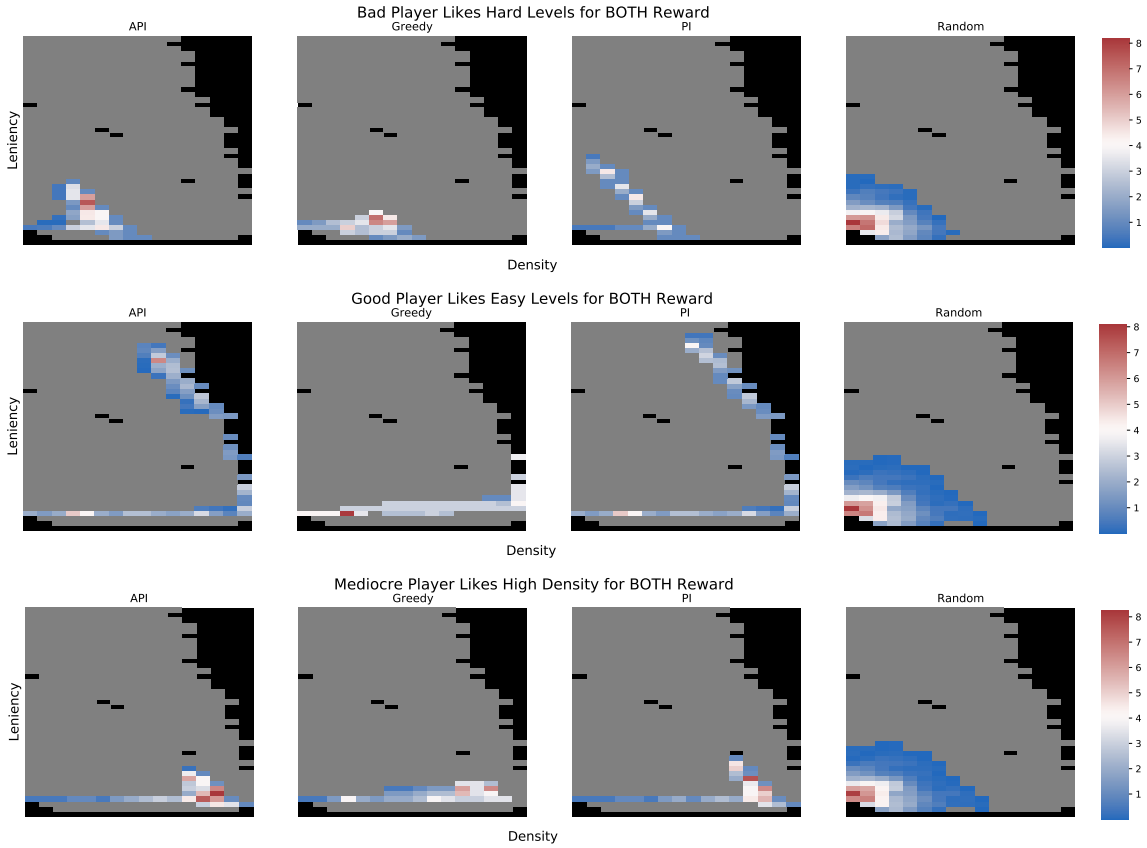


Figure 5.4: Heat map of cells in the MAP-Elites grid visited by each director for *Icarus* segment generation.

5.4.3 Evaluation

Evaluation is done in two parts. The first part is similar to the first case study, in which level segments are generated, and the directors are compared. The second part examines what happens if there is a player proxy switch, starting with *Good Player Likes Hard Levels* and switching to *Bad Player Likes Easy Levels*.

5.4.3.1 Segment Generation

To test segment generation, each director was run twenty times on different seeds. Fifty levels were generated for *Icarus* per run. A level was made up of five states; there could be up to four additional linking states. Figure 5.5 shows sample levels generated. The results are in Table 5.3. Figure 5.4 shows heat maps for three players based on the average number of times a player visited a cell; the highest value of R_D is at the top right of the heat map.

Player	Director	Reward	Percent Complete
Bad Player	API	$\mathbf{0.1698 \pm 0.0767}$	0.6555 ± 0.2986
Likes Easy	PI	0.1522 ± 0.0912	0.5594 ± 0.3368
Levels	Greedy	0.1178 ± 0.0739	0.4046 ± 0.2770
	Random	0.1208 ± 0.0814	$\mathbf{0.9884 \pm 0.0808}$
Good Player	API	$\mathbf{0.5079 \pm 0.2486}$	0.8442 ± 0.2683
Likes Easy	PI	0.4826 ± 0.2519	0.7870 ± 0.3043
Levels	Greedy	0.4965 ± 0.2216	$\mathbf{1.0000 \pm 0.0000}$
	Random	0.5009 ± 0.2143	$\mathbf{1.0000 \pm 0.0000}$
Good Player	API	$\mathbf{0.6093 \pm 0.3150}$	0.9217 ± 0.2053
Likes Hard	PI	0.5751 ± 0.3070	0.8639 ± 0.2716
Levels	Greedy	0.4519 ± 0.2006	$\mathbf{1.0000 \pm 0.0000}$
	Random	0.1323 ± 0.0972	$\mathbf{1.0000 \pm 0.0000}$
Mediocre	API	$\mathbf{0.4427 \pm 0.2320}$	0.76840 ± 0.2964
Player Likes	PI	0.3974 ± 0.2183	0.6278 ± 0.3028
High Density	Greedy	0.4187 ± 0.2140	0.6847 ± 0.3389
	Random	0.1375 ± 0.1205	$\mathbf{1.0000 \pm 0.0000}$
Mediocre	API	$\mathbf{0.2835 \pm 0.1366}$	0.7089 ± 0.2886
Player Likes	PI	0.2112 ± 0.1422	0.4494 ± 0.3131
High Leniency	Greedy	0.1950 ± 0.1015	0.6609 ± 0.3229
	Random	0.1272 ± 0.0858	$\mathbf{1.0000 \pm 0.0000}$
	API	$\mathbf{0.4026 \pm 0.2694}$	0.7797 ± 0.2895
All Players	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 5.3: Average plus or minus the standard deviation of the reward and percentage completed for each director for Case Study 2 for *Icarus* level assembly.

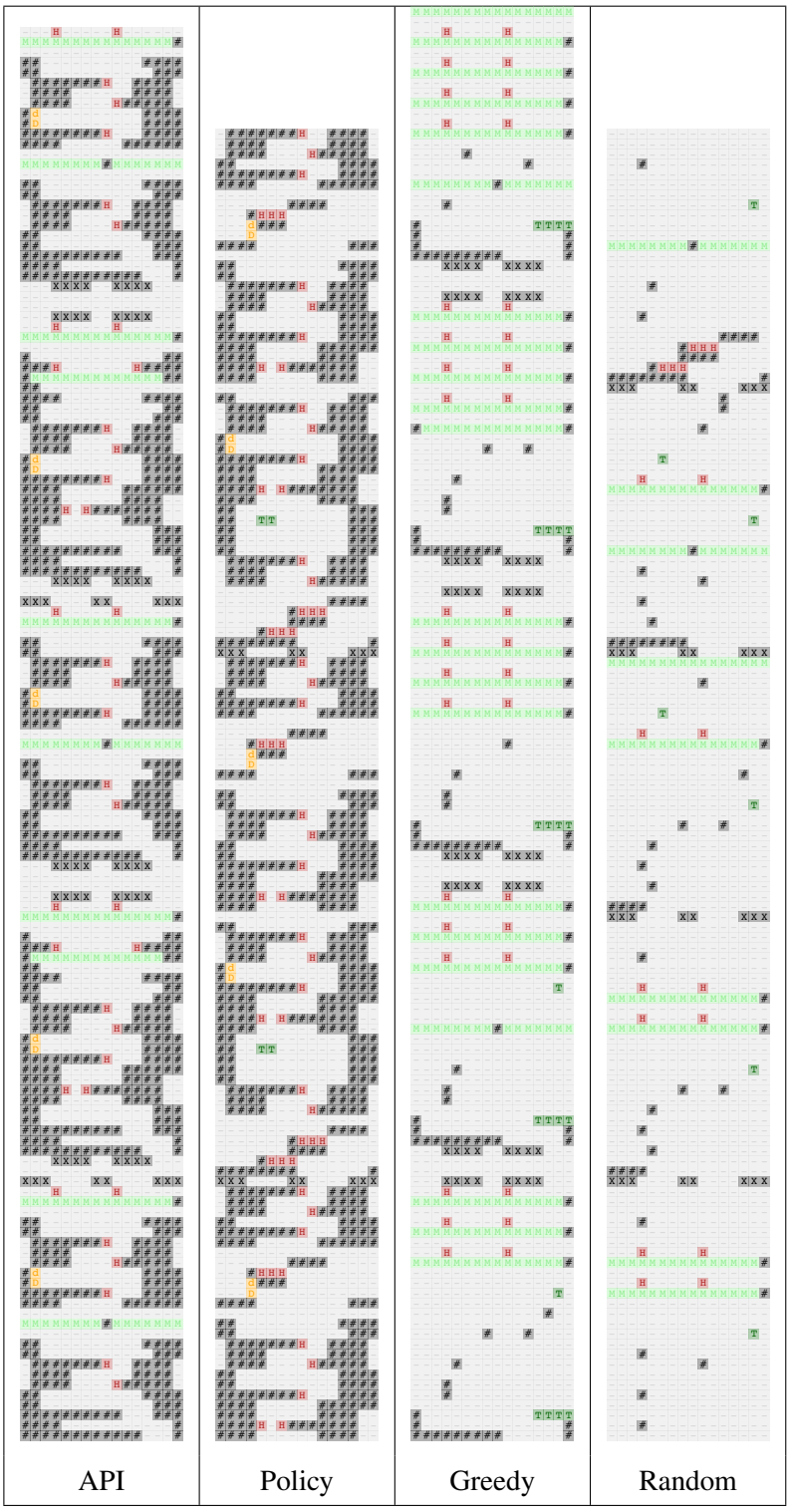


Figure 5.5: The first and last level generator by each director for *Icarus* with *Good Player Likes Hard Levels* for *Icarus*.

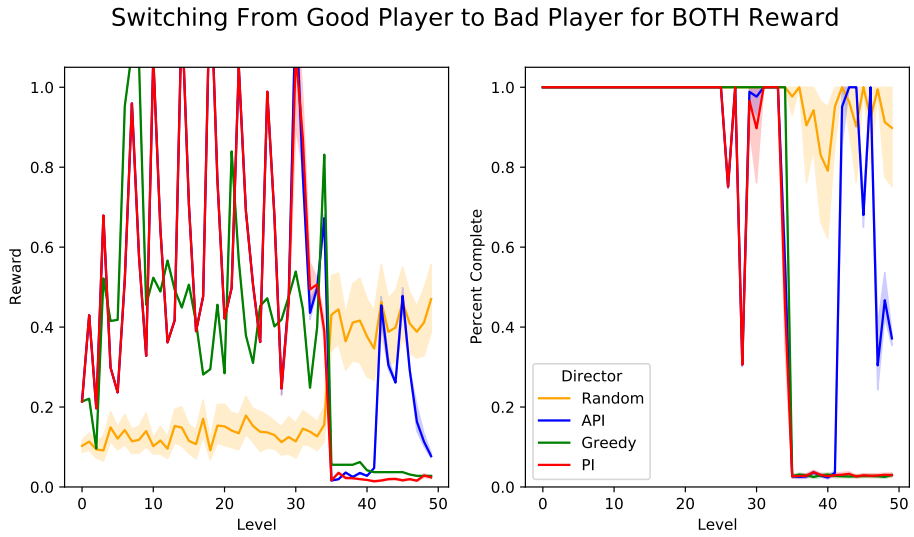


Figure 5.6: Shows the average reward per level (top) and the average percent complete average per level (bottom) when the player is switched after 35 levels from *Good Player Likes Hard Levels* to *Bad Player Likes Easy Levels*.

Starting with the average reward, API was the best performer for all player proxies. PI performed second best for all proxies except *Good Player Likes Easy Levels*, where it was the worst performer. The reason is shown in the middle plot of Figure 5.4. Both API and PI quickly moved to the area with the largest designer reward. Because API reduced failure, it was able to get a larger reward, while PI continually forced the player to try and fail difficult levels. This phenomenon repeats for *Bad Player Likes Hard Levels* and *Mediocre Player Likes High Density*: PI over-explored the failure front, whereas API found the front and moved backward.

Moving to percent complete, the cost of finding levels that the player failed was seemingly worse results. However, this was not necessarily problematic. After all, a game where the player never lost would be boring [62]. Random was consistently the best for percent complete across all players since it stayed near the origin of the graph. We also see one of the failure points for Greedy on *Good Player Likes Hard Levels* where it could not explore the space enough to find a difficult level for the player. API and PI did not have this problem.

5.4.3.2 Switching Player Proxies

A potential pitfall with systems that adapt to a player is the problem of switching players (e.g., one person is playing, and then they hand the controller over to a friend). A system that fits

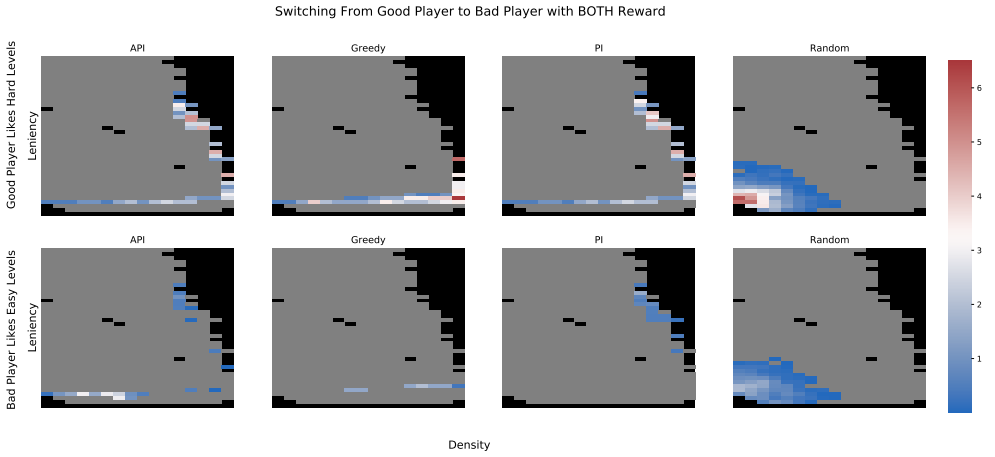


Figure 5.7: Shows the areas explored by each director. Top graph is for the first 35 levels when *Good Player Likes Hard Levels* is playing and the bottom graph shows when *Bad Player Likes Easy Levels* takes over.

a player may only work for that player. When a player switch occurs, the director should quickly adapt to the new player. A test was run to see how the system works on this problem.

Good Player Likes Hard Levels played for 35 levels and then switched to *Bad Player Likes Easy Levels* for 15 levels. These two particular proxies were chosen because the former was an ideal player proxy given the initial reward function, and the latter was not. When the switch occurs, the MDP will favor harder levels that *Bad Player Likes Easy Levels* cannot beat and does not enjoy. Each director was run twenty times. The directors did not receive notice of the player proxy change.

The results are in Figure 5.6, and a heatmap of the levels played by both proxies are in FIgure 5.7. Looking at Figure 5.6, API and PI are identical at the start, 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 went up since it happened to stay near the easy levels that the new player liked. From level 35 to 50, Greedy and PI showed no signs of adapting. API adapted after about 7 levels and produced playable levels for the new player proxy. This can be seen in Figure 5.7 where levels to the bottom left were played by *Bad Player Likes Easy Levels*.

5.5 Limitations

- The main limitation of the work presented in this Chapter is the use of agents to validate whether the approach works. The only way to truly know is to test with people. This will be addressed in Part III.

- The assumption that BCs can be used to assess difficulty, enjoyment, or anything other than exactly what they claim to measure is a faulty assumption—for example, density measures density, not difficulty. This is addressed in Chapter 7.

- Linking states were included in the MDP due to the off-chance that failure occurred during them rather than the states. This was not the best decision. A cleaner implementation would have been to include links in the edges. Linkers as a state makes the implementation more complex, requires more memory, and slows solving the MDP. The problem of player failure while moving through a link can be resolved by treating the link as part of either segment. This is implemented later in Chapter 8.

5.6 Unexplored Areas of Future Work

- The problem of repeating states could have been solved but was not tested. The process theorized was to select a state, update the policy assuming player success, and repeat until the full level was made. This makes repeating patterns less likely because $N(s)$ would be updated with each step, reducing the reward of states in those patterns. However, this would likely be slow, especially for case study 1, requiring many MDP updates. A way around this could be to frame it as an infinite generator where segments are appended to the level as the player gets close.

- One point worth considering for the random director is that work has already been done in mathematics to determine how it will behave [168]. The *random* director is effectively a random walk agent. Therefore, given a 1-dimensional or a 2-dimensional space, the agent will act recurrently. Given an infinite time horizon, this means that the agent is guaranteed to return to the origin state. The agent may never return to the origin if given a higher-dimensional space ($\geq$ 3). If the previous point on repeating states were implemented, it would be interesting to compare that exploration heatmap versus the results in 5.4.

- *Spelunky* has a level generator that mixes handmade segments with procedural design [210]. Instead of a pseudorandom selection of handmade segments, an MDP could be built on top to select segments.

- A multi-objective MDP [52, 46] with a finite horizon [178] could better model the problem space. The finite horizon would lead to a more accurate utility calculation for the domain of level assembly. Multi-objective optimization would reduce the complexity of reward balancing and let the designer focus more on their game.

- The published paper of this chapter has a version of the following as an area for future work: "By incrementing $N(s)$ for all states associated with a cell in the MAP-Elites grid, the goal was to make it so the user was not forced to play similar levels. This was trying to address the '1000 bowls of oatmeal problem,' which is the phenomenon of generating computationally distinct content that is indistinguishable to players [32]. Another way that would have simplified the construction of the MDP, reduced the complexity of updates, and reduced the size of the MDP would have been to treat each state as a cell. Therefore, each state would have contained multiple elites. When getting a level segment from the state, it could be simple, where a state returns a random elite. This could be improved with the state returning an elite based on a weighted random sampling based on fitness. Further, it could track which level segments were previously played to guarantee that the same level segment was not repeated until the player played every other elite." However, incorrect assumptions were made around how the digraph is affected by multiple elites per node, and the problem is fully discussed in Section 6.4.

5.7 Conclusion

The solution presented for **RQ1** was to formulate the digraph built with Gram-Elites and linking as a MDP. The rewards were built from heuristics. Heuristics as a reward have already been discussed as a limitation, and it is addressed in the next chapter.

N-gram generation was difficult for every director because using n-grams to form an MDP is an unideal use case. The first problem was that assembled levels were not guaranteed to be completable. Even once a completable level was found, reward decay encouraged the director to find a new level, which resulted in the agent playing more levels that were not completable. The second problem was that n-gram generation required the policy to select many level slices to create

a single level. This resulted in a repeating pattern of level slices, as seen in Figure 5.2. Using level segments helps address the problem of looping because the policy is used much less; repeated patterns only become a problem if the player makes it to the area with the largest reward, in which case the player has ostensibly beaten the game.

Alternatively, level assembly from level segments was a better use case. The API director outperformed the other directors for every player type. Further, API was the only director that adapted to the simulated real-life scenario of a player giving their controller to a friend.

This marks the end of answering **RQ1**: How can usable levels be assembled from level segments for dynamic difficulty adjustment? The answer is shown visually in Figure 1.4. The most important step is by far the last one, which is to use an MDP to figure out which level segments should be used to form a larger level for the player. Another approach, Q-learning [199], was tested in early work, but the problem was that a real player would need to play tens, if not hundreds, of levels for Q-learning to work well. If this dissertation had ended here, Q-learning would have been an acceptable answer. However, **RQ2** takes us from working with agents to working with players, and asking a player to play hundreds of levels before the system can work is not reasonable. Before we get to **RQ2**, though, there is one final chapter in this part of the dissertation which discusses *Ponos*, the name of the software built that puts everything from **RQ1** together.

Chapter 6

Ponos

> And he remembered his money, his shop, his house, the buying and selling, and Mironov's millions, and it was hard for him to understand why that man, called Vasili Brekhunov, had troubled himself with all those things with which he had been troubled.
>
> Leo Tolstoy, *Master and Man*

This Chapter is different from all the others that have come before because it is about how you can use everything that was described in the previous three chapters. Gram-Elites and linking were combined into one software repository called *Ponos*.[1] This chapter describes (1) how you can use *Ponos*, (2) what the output of *Ponos* is, (3) how you can use that output to make an MDP for level assembly, and (4) why multiple elites per node in the digraph is a bad idea.

Before getting into those four topics, though, I want to take a moment to explain why I chose the name "*Ponos*." It is, to be perfectly candid, a name that I do not like. It falls into what has become a trope for software names, which is to be named after a Greek deity, in this case, Ponos. Ponos is said to be the personification of toil and stress. I thought that was a funny name for a software system whose goal is DDA via PLG. Unfortunately, in all my reading, I have not encountered Ponos, so my understanding of Ponos is surface-level at best; I imagine that the form of toil that Ponos represents would not be viewed positively. Perhaps a better name would have

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

been something related to Sisyphus and the impossible "level" Zeus gave him, but the goal is not to generate impossible levels. The goal is for the player to experience challenge, or to toil and stress until completing a level.

6.1 How Do I Use *Ponos*?

While coding Gram-Elites and linking, my goal was not to generate a system that others could use. I was focused on generating levels and putting them together to make larger levels. For *Ponos*, though, there was a different kind of question: how do I make this system easy to use for someone else? The first part of the answer was simple: combine the codebases for Gram-Elites and linking so that the developer only has to figure out how to work with one repo rather than two.

There was a bigger problem: *Ponos* needed to be able to interact with the developer's game. For example, it needed to be able to build a level and then ask, "How completable is this level?" The code for Gram-Elites and linking was written in Python, but most games are not written in Python. Communication between *Ponos* and a game not written in Python would be an annoying problem for developers. For example, it is unreasonable to ask developers to reimplement their game in Python if they want to use *Ponos*. An alternative is to ask developers to create a binding with an API. However, having worked on bindings before, I knew this was a pain, so I dismissed it.

Another approach is to make an executable that can be used as a subprocess.[2] This approach was taken early on when working on Gram-Elites for *Mario* by using the *Mario AI Framework* [185]. Levels were passed to the subprocess via a named pipe, and it eventually returned how completable a level was. This is a fine approach, but I would caution anyone to avoid starting a subprocess every time a level segment is going to be tested. I didn't go this route because I was unsure if named pipes would work on Windows, and the alternative of a file-based communication between processes was unappealing to me.

This leaves one alternative, which is a server-based approach. To work with *Ponos*, the developer has to implement a server (REST or socket-based, I recommend socket-based due to REST being slower), and that server has a set of requirements which will be described in a moment. The server does not have to be hosted on some service like AWS; it can run locally. *Ponos* will work if someone wants to host their server. The reason why I think that this is the easiest approach is that anyone who can write a game can write a simple server.

[2] https://docs.python.org/3/library/subprocess.html

The required endpoints for any server written for *Ponos* are:[3]

- *config* - Return a configuration for *Ponos*. The configuration is described in Section 6.1.1.
- *levels* - Return a list of input levels that will be used by Gram-Elites.
- *assess* - Receive a level and return an assessment of said level. The assessment should include (1) percent completable and (2) an assessment of the level based on every relevant BC.
- *reward* - Receive a level and return the reward for said level.

The most complicated part is implementing an agent for your game. This is because (1) some games are not initially engineered with this consideration, and it can be a difficult transition, and (2) the agent is the bottleneck for *Ponos*. If the agent is fast, then *Ponos* will be fast; if the agent is slow, then *Ponos* will be slow. For example, imagine it takes a second to test a level, then with the configurations from Chapter 3, it would take 500 seconds to generate the initial population and $100,000$ seconds to run all the iterations; double $50,000$ because crossover produces two levels, not one. In total, you get $\approx 100,500$ seconds, or $\approx 1,675$ minutes, or ≈ 27.916 hours for *Ponos* to run. This leaves little time for iteration, which, as will be discussed in Chapter 10, is essential.

Once these endpoints have been implemented, using *Ponos* is simple. First, you start your server. Then you run *Ponos* with a command like: `pypy3 ponos/ponos.py --port 8080 --model-name ponos-works.json`. There is a command-line option to use REST instead, but the default is to use a socket.

6.1.1 *Ponos* Config

- *seed* - the seed for the random number generator.
- *death reward* - reward for the *death* node. It should be something negative unless the goal is to make the player lose every level they play.
- *end reward* - reward for the *end* node. Can be positive or negative.
- *n* - this is the size of n in the n-gram used by Gram-Elites.
- *levels-are-horizontal* - a boolean for whether levels are horizontal. Set to `false` for games like *Kid Icarus*.

[3]There is also an example repo for using *Ponos* with *Mario*: https://github.com/bi3mer/ponos-example

- *start-population-size* - the number of levels generated by n-grams for initial population generation in Gram-Elites.
- *iterations* - number of iterations that Gram-Elites runs for when running n-gram-based genetic operators.
- *start-strand-size* - the size of levels generated by n-grams.
- *max-strand-size* - how large a level can become when running n-gram-based genetic operators.
- *mutation-rate* - likelihood of mutation.
- *structure-chars* - an array of characters that designate a structure (e.g., `['[', ']', '<', '>']` for *Mario*).
- *custom-linking-columns* - an array of custom linking columns. Can also not be provided and *Ponos* will find the columns itself based on the *structure-chars*. However, I recommend defining your own custom linking columns so that *Ponos* will run much faster.
- *structure-size* - the size of a structure in your game.
- *allow-empty-link* - a boolean for whether the linking algorithm can return an empty link.
- *max-linker-length* - the longest that a linker can be before the linking algorithm should give up.
- *computational-metrics* - an array of computational metrics with the following details:
  - *name* - the name of the metric, this will be used when the *assess* server call is used.
  - *min* - the minimum return value for the metric.
  - *max* - the maximum return value for the metric.
  - *resolution* - the number of bins for the metric.

6.2 What Does *Ponos* Return?

Ponos returns a JSON file with the name provided by the *–model-name* command line argument. In the example command from above (`pypy3 ponos/ponos.py --port 8080`

`--model-name ponos-works`), the output will go to the file `ponos-works.json` in the local directory of *Ponos*.

The JSON file contains a dictionary with two keys: `nodes` and `edges`. The `nodes` key links to an array of dictionaries. Each element in the array has the following elements:

- *name* - the node's name, which is based on its BCs (e.g., 0_0 would be a node with the BCs of 0 and 0 for a configuration that specified two BCs via `computational-metrics`).
- *reward* - a float that was assigned by the game server's *reward* call.
- *is-terminal* - a boolean for whether the node is terminal or not. Only the *end* and *death* nodes should be terminal.
- *level* - an array of the level segment generated by Gram-Elites broken down into level slices.[4]
- *depth* - the distance from the *start* node to the given node based on the edges.

The `edges` key also links to an array of dictionaries. Each dictionary has the following elements:

- *src* - the name of the node where the edge starts from.
- *tgt* - the name of the node where the edge goes to.
- *probability* - an array that lists the probability that the player reaches the target state or somewhere else. This is represented as an array of arrays (e.g., you may have an edge from 0_0 to 0_1 with the probabilities like: `[["0_1", 0.99], ["death", 0.01]]`).
- *link* - an array of level slices that may be empty, depending on the level segments being linked and whether the configuration allows for an empty linker.

6.2.1 Start and End Node

When you run *Ponos*, it will output some basic stats to keep you updated. Here is a sample output from one of the runs I did while testing BCs for *DungeonGrams* in Chapter 9:

[4] If a level has *is-terminal* set to `true`, then there does not have to be a level associated with it.

```
Result will be stored at: dg.json
SOCKET
Running MAP-Elites...
Level Generation: population generation...
Percent [####################] 100% Done
Level Generation: segment optimization...
Percent [####################] 100% Done
Adding nodes...
Built 2472 level segments.
Linking edges...
Possible links: 7768
Links made: 7682/7768
Nodes left: 1146
Max depth: 19
Adding edge: 18_0_0_0-0 -> 'end'
Adding edge: 16_0_0_2-0 -> 'end'
Adding edge: 16_2_1_0-0 -> 'end'
Adding edge: 18_1_0_0-0 -> 'end'
        First:  0_0_0_0-0
        Last:   ['18_0_0_0-0', '16_0_0_2-0', '16_2_1_0-0', '18_1_0_0-0']
Nodes removed: 41
Updating node rewards...
Percent [####################] 100% Done
DONE
Time to run: 1136.6960000991821
```

The number of level segments made for this example was $2,472$, but the number of nodes left after linking was $1,146$. In other words, we lost $1,326$ level segments in the linking process, even though almost every possible link was successfully made. A reasonable question is, why? Chapter 10 will discuss this in more detail based on the curse of dimensionality as related to BCs, but it is also related to how *Ponos* finds the *start* and *end* node.

One way that *Ponos* could work is to find the largest sub-tree and use that tree to find the *start* and *end node*, but it does not do that even if more level segments could be saved. Instead,

Ponos selects the node with the lowest sum of BCs to connect to the *start* node.[5] This is because a level progression that starts with level segments that are too complicated is useless. It won't matter how many levels there are if the player cannot beat the first level. So, *Ponos* selects for the simplest levels, based on BCs, to connect to the *start* node.

Alternatively, the segments that should connect to the *end* node are found through a breadth-first search, starting from the *start* node, which has been connected to the already found nodes with the lowest sum of BCs. The search is conducted to find the maximum depth nodes. These nodes are connected to the *end node*. The sum of a node's BCs is not directly relevant. In the example above, all four segments had the same depth.

Any node that cannot be reached from the *start node* and cannot reach the *end* node is removed during this process.

6.3 How Can I Use the Output of *Ponos*?

There are two ways that you can use the output JSON file. The first option is to use a tool that I've built called GDM, which stands for graph-based decision making, and it is a codebase written in Python[6] and TypeScript.[7] The tool allows users to create a graph-based MDP and run policy iteration on it to build a policy that can be used for, in this case, level assembly. Neither version has a functionality built in to read the JSON file from *Ponos*, but doing so is fairly simple. I have also written a script that converts the output file to TypeScript. I also like to change the output of *Ponos* to include an ending level segment that I have hand-defined.

The other option is to write some code that reads the graph file and converts it into an MDP. The code will also have to implement policy iteration to get a policy from the MDP. Games not written in Python or TypeScript could approach this by creating a binding to GDM, but I would not recommend it. The code is simple and not hard to redo in another language, as I learned when I rewrote GDM from Python to TypeScript.[8]

GDM does not come with a version of API built into it. This, though, is fairly simple, and there are examples online.[9] The implementation amounts to checking if the player lost, and removing a number of edges from the graph based on how many times they have lost in a row.

[5] There can be multiple nodes that are connected to the start node if they all have the same sum of BCs.
[6] Code available on GitHub: https://github.com/bi3mer/GDM
[7] Code available on GitHub: https://github.com/bi3mer/GDM-TS
[8] I did this for the online player studies which were part of answering **RQ2**.
[9] For example: https://github.com/bi3mer/recformer/blob/main/src/LevelGeneration/mdpLevelDirector.ts#L92

6.4 A Note on the Linking Implementation

Before linking is run, every Gram-Elites bin with at least one elite is turned into a node in a digraph. A tempting implementation is to make one node that contains all the level segments. Then, when a policy selects that node, a random level segment can be selected. It's a relatively clean implementation, but there is a problem with this approach.

For example, suppose that we have three nodes: A, B, and C. Each node has three elites: A_1, A_2, A_3, B_1, etc. Linking can work by generating links for each elite, such as a link from A_1 to B_1, B_2, and B_3. If at least one link is found, then an edge can be added from A to B. Let's say that the ordering for linking also generates edges from B to C: $A \rightarrow B \rightarrow C$. If the goal is to generate a level that is three segments long, there can be an issue. Say that linking finds a link from A_1 to B_1, but fails for the other eight possible links. Generating would always start A_1 because there would be no other option. The next level segment used would have to be B_1. At this point, the level is composed of level segments $[A_1, B_1]$. One more link to go, but what if no outgoing link was found for B_1, but there were outgoing links for B_2 and B_3, so there was an edge from B to C. The policy for level assembly wouldn't work. There are ways to solve this, but the easiest solution is not to have nodes with multiple-level segments. Therefore, *Ponos* separates every node into multiple nodes, based on the number of usable elites, and then runs linking.

6.5 Conclusion

In conclusion, *Ponos* is a tool that puts together Gram-Elites and linkers. For a designer to use it, they have to build a server for their game that *Ponos* can call for basic information like a configuration and how completable a level is. This design allows *Ponos* to work for games that are not just built in Python. *Ponos* outputs a graph that can be used to build an MDP which can assemble levels for DDA.

Part III

Research Question 2

Chapter 7

Heuristics for Predicting Difficulty and Enjoyment

> "He said that one could only stake four thousand florins at once, didn't he? Come, take it, stake the whole four thousand on red," Granny commanded.
>
> Fyodor Dostoyevsky, *The Gambler*

This Chapter is based on the paper, "Solution Path Heuristics for Predicting Difficulty and Enjoyment Ratings of Roguelike Level Segments" [14]. The authors are me and Seth Cooper.

This Chapter addresses **RQ2.1**: How well do computational metrics approximate difficulty and enjoyment? This question is purposefully broad. The only way to definitively say, "No, heuristics can not be used to approximate difficulty and enjoyment" would be to test every possible heuristic exhaustively. Therefore, the answer to the question will be based on a selection of heuristics but with the caveat that yet-to-be-discovered heuristics could change the answer.

RQ2.1 is an important question because creating enjoyable and challenging levels poses a significant challenge in level generation. As a result, there has been a plethora of work looking at dynamic difficulty adjustment [84, 214], experience-driven games [159, 83], and more. Methods that automatically and accurately assess content could greatly improve PCG methods.

Heuristics can be avoided, though. An alternative approach to heuristics for guiding level generation is to use player modeling [208]. By developing a model that finds what a player likes and dislikes, player modeling can guide a PCG system to build levels custom-tailored to the user

[90]. The problem is that the player model must quickly and accurately model the user to be usable.

As a result, this Chapter focuses on heuristics rather than player modeling. A study with 143 players on *DungeonGrams* was run. Players were asked to rate levels on a 7-point Likert scale based on difficulty and enjoyment. Separately, levels were analyzed based on eleven heuristics. These heuristics used the initial level, the solved level, and the path found by an agent to solve the level. The output of these heuristics was used as input for a linear regression model. An ablation study, using the player ratings as the true values, determined which heuristics were most useful in approximating difficulty and enjoyment.

Heuristics based on solution paths were the most useful for predicting the difficulty and enjoyment of a level. A heuristic that uses Jaccard similarity—see Section 7.2.1—was highly effective in predicting the difficulty of a level. This finding stands out because no previous work, as far as we are aware, has used Jaccard similarity of paths for game research in this way. Jaccard similarity as a heuristic played a less prominent role in predicting how enjoyable a level was. Instead, the most important heuristic was one that used the path to solve the level and looked for enemies nearby. This heuristic was also used prominently in predicting difficulty. However, there was no combination of the used heuristics as input into a linear regression model which outperformed a simple baseline for predicting enjoyment.

Surprisingly, there was little, if any, correlation between the degree of difficulty and enjoyment experienced by the player [141].

7.1 Related Work

Gellel and Penny [59] calculated the "interestingness" of a level as the difference between the length of the solution path and the length of the solution path if there were no obstacles—locks, keys, enemies, etc. The larger the difference, the more "interesting" the level because the player is required to explore more to complete the level.

Evaluating a roguelike level for its difficulty is typically based on solution paths. Weeks and Davis [200] used the same approach except on a layout of rooms and hallways to estimate the difficulty of a layout. Sampaio et al. [148] used a different approach where, given a desired difficulty, required items were placed into a dungeon based on the distance to the player's starting point and the nearby entities to the placement point. Unfortunately, these studies did not validate their heuristics with a player validation study.

Solution paths are not the only way to approach difficulty. Gonzalez-Duque et al. built an adaptive system to predict the time it will take a specific player to beat a level [62], specifically for a roguelike game and Sudoku. Time is an approximation for difficulty in puzzle games in that it can be considered that the longer the player takes to solve a level, the harder the level. In their work, they evaluated a Bayesian optimization algorithm by running a user study and found positive results when compared to other baseline approaches.

A different approach to creating heuristics by hand is to use crowd-sourced data. Jennings-Teats et al. ran a study where users were shown short level segments of a platformer and asked to classify difficulty between one (easy) and six (hard) [90]. With this data, they trained a classifier to rank the difficulty of any input level. Reis et al. took a different approach [141]. In their work, they crowd-sourced difficulty evaluation of all the level segments for a platformer. One interesting point is that Reis et al. found a high correlation between difficulty and enjoyment, which was not replicated in the results found in this chapter—this is examined more in Section 7.3.3, but note that a roguelike is used in this chapter and Reis et al. used a platformer, and the difference in game genre could account for the different results.

Mariño et al. ran a study on metrics for estimating difficulty, enjoyment, and visual aesthetics of *Mario* levels on a 7-point Likert scale [117]. They found that the computational metrics they used (linearity [162], leniency [162], density, and compression distance [152]) could not accurately predict enjoyment and could be misleading when estimating difficulty. They concluded that computational metrics could not replace a well-designed user study. While this chapter uses a roguelike instead of a platformer, it was also found that the heuristics tested poorly estimated enjoyment. When it comes to difficulty, a combination of heuristics can provide a strong estimation of difficulty, but the only way a combination was found was through a user study.

Wang et al. created a method designed to be game agnostic in the sense that it could estimate difficulty for any puzzle game that can be solved by a solver [198], such as Clingo [58]. They used the results of the solver to rate difficulty based on the number of required solver calculations, guesses, backtraces, certain branches, uncertain branches, and the ratio of certain to uncertain branches. These variables can be broken down into three categories: *initial features* (features related to the start state of the puzzle), *solution features* (features related to the end state of the puzzle), and *dynamic features* (features related to the solution of the puzzle) [193]. These six variables were fed into a GA to find a formula that best matched a training set of Sudoku puzzles to difficulty. They found that the number of uncertain branches was highly correlated to difficulty. A different approach is to use weighted linear regression instead of a GA [193].

Another area of work is the study of chess puzzle difficulty [70, 82]. Of note in this area are websites like Chess.com and Lichess.org, which have many players playing chess puzzles. With so many players, there is no need to automate difficulty evaluation. Instead, these websites can actively update puzzle ratings—referring to the expected player rating needed to solve a puzzle—based on player performance and update real-time [61, 150]. Lichess keeps an open database of puzzles with ratings that could be used in future research to validate generic puzzle difficulty approaches.[1]

7.2 Method

Unlike the work in the previous chapters, this Chapter only examines solution path heuristics for a single game: *DungeonGrams*. This is because the focus of this Part of the dissertation is evaluating the method with *DungeonGrams*, a roguelike. The next Part of this dissertation will examine how the method works with a platformer and takes an approach that is different from the one taken in this Part.

7.2.1 Heuristics

Heuristics used initial features, solution features, and dynamic features [193]. The A* implementation built for DungeonGrams in the previous chapters was used to find a path.[2] That path was used to obtain the solution features and dynamic features of the level.

The downside of using A* is the lack of access to information like backtracking and uncertain branches [198, 193]. The benefit is that the work may be more accessible because building a tree search for a game is easier than re-implementing a game as a logic program.

A* was run three times for each level. The first time was on the original level. The second was on a version of the level where enemies were removed. The third ran on a version of the level without enemies and switches.

Seven heuristics evaluated are based on work discussed in Section 7.1. Two heuristics—linearity and leniency—are from the work of Smith and Whitehead [162]. Finally, two new heuristics were built with Jaccard similarity [86]. In total, eleven heuristics were evaluated.

[1] https://database.lichess.org/#puzzles

[2] A path, in this case, being an array of tile coordinates (x, y).

7.2.1.1 Jaccard Similarity

$$\text{JaccardSimilarity(A,B)} = \frac{|A \cap B|}{|A \cup B|} \tag{7.1}$$

Jaccard similarity [86], see equation 7.1, is the size of the intersection of two sets divided by the size of the union. In terms of paths, it is the number of common points between two paths divided by the number of unique points for both paths.

The motivation for introducing and testing Jaccard similarity as a heuristic for difficulty and enjoyment comes from the observation that path length difference does not capture enough of the differences between two paths. For example, if there are two paths P_a—path to beat a level—and P_b—path to beat the level, but the level was modified so that all enemies were removed—and $|P_a| = |P_b|$. The path length difference ($|P_a| - |P_b| = 0$) implies that the two paths are the same. As a heuristic for this example, path length difference says that the two paths are equivalent. But, what if $P_a \cap P_b = \emptyset$? Since there are no common points between the two paths, it is wrong to say that the paths are 100% similar. From this line of reasoning, the hypothesis was that Jaccard Similarity would be better suited as a heuristic for estimating difficulty and enjoyment because it better captures differences in solution paths.

7.2.1.2 Evaluated Heuristics

- **path-no-enemies** - Difference between the path length of the original level and the path length of the level with no enemies.
- **path-nothing** - Difference between the path length of the original level and the path length of the level with no enemies and no switches.
- **jaccard-no-enemies** - Jaccard similarity of the path of the original level and the path of the level with no enemies.
- **jacard-nothing** - Jaccard similarity of the path of the original level and the path of the level with no enemies and no switches.
- **proximity-to-enemies** - Using the completion path found for the original level, each position in the path was used to search for enemies up to three tiles away in every direction. For each enemy, one over the Manhattan distance to that enemy from the tile on the path was summed. This means that the more enemies near the optimal completion path, the larger the

value of this heuristic. The sum was then divided by the length of the path. This is similar to a heuristic from Tremblay et al., which uses the exact distance for every agent via an A* search to calculate a distance to enemy metric [187].

- **proximity-to-food** - The same as *proximity-to-enemies*, except it searches for food rather than enemies.
- **stamina-percent-enemies** - Percent difference between the ending stamina for the tree search on the original level and the stamina left for the level with no enemies.
- **stamina-percent-nothing** - Percent difference between the ending stamina for the tree search on the original level and the stamina left for the level with no enemies and no switches.
- **density** [162] - Number of solid tiles, including spikes, divided by the total number of tiles in the level.
- **leniency** [162] - Number of enemies, spikes, and switches divided by the total number of tiles in the level.
- **food-density** - Number of food tiles divided by the total number of tiles in the level.

7.2.2 Level Dataset

Level segments generated by Gram-Elites in Chapter 3 for *DungeonGrams* were used as a level dataset. However, not every level generated was used in this study. The original work generated multiple elites per cell in the MAP-Elites grid. The problem was that elites in the same cell ended up being very similar, to the point where it was questionable whether players would be able to tell the difference [31]. Therefore, rather than potentially waste resources and evaluate every level segment, it was decided that there would only be one level segment per cell. This filtering was done by random selection. This was possible because a filtering process had already been run to remove level segments that were not usable. This left a total of 191 levels.

7.2.3 Player Study

An online study that recruited 150 workers was run on Mechanical Turk.[3] Methods had approval from Northeastern's Institutional Review Board.

[3] https://www.mturk.com/

Each participant was paid 1 dollar. Hourly wage was estimated by tracking a user's time to beat a level, allowing for a maximum of 60 seconds per attempt. Based on actual playtimes after running the study, a bonus of 50 cents was added for an estimated median hourly wage of twelve dollars an hour.

Each participant was asked to agree to a consent form and then taken to the game. At the game, they were immediately presented with a payment code so they could receive payment without playing a single level. For those who opted to play, they first played a tutorial level that helped show the basics of how to play the game. If the player lost a level twice, they were given the option to give up and skip the level to avoid players becoming frustrated and quitting.

After they beat the tutorial level or gave up, a questionnaire appeared with two statements: "This level was difficult" and "This level was enjoyable to play." The player could respond by selecting between "strongly disagree" and "strongly agree" on a 7-point Likert scale, with the middle being "neutral".

After the survey, a random level was selected from the level dataset for the player to play. The survey followed after they completed the level. This continued until the player hit 11 levels played,[4] including the tutorial level, or they quit. No identifying information was requested or stored.

7.2.4 Predicting Difficulty and Enjoyment

The heuristics from Section 7.2.1 were calculated for every level and made into a dataset with median player-rated difficulty and median player-rated enjoyment. Note that the expected output was a continuous value instead of categorical. The reasoning for this decision is as follows. There are multiple player responses for each level. One solution to take advantage of these would be to make the correct output for each level be the mode. However, there is an ordering factor implicit in a rating scale, and it was important to capture that not only for this work but also for the work in the next Chapter. As a result, converting the survey results to continuous values was best for this work. Therefore, the median was used instead of the mean because it better represents the ordinal data from a Likert survey [209], especially when the dataset size is small [171].

To evaluate which heuristics were important, an ablation study was run using the training portion of the data, where the dataset was split with a train-test split of 0.8. A linear regression model was used [145]. Due to the limited size of the dataset, k-fold cross-validation was used,

[4] Based on a pilot study, the average time spent per level was calculated. The average time was used to estimate costs and came up with eleven as the maximum number of levels a player can play and still be compensated fairly.

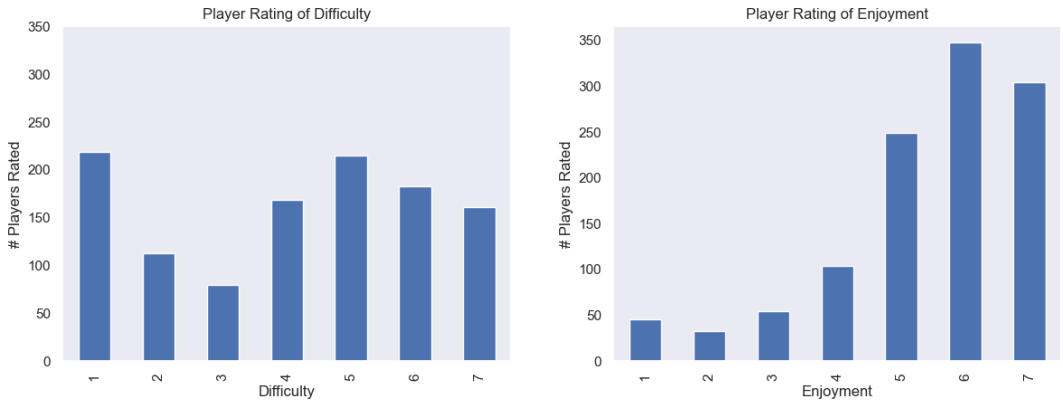


Figure 7.1: Survey results from players ranking difficulty and enjoyment for every level they played.

where $k = 5$, and minimized the mean square error (MSE). The tutorial level was removed from the dataset due to being an outlier, likely a result of being the first level played. The top ten combinations out of $2,047$ potentials were used to evaluate feature importance.[5] The best combination of heuristic features—as defined by the combination of heuristics, which resulted in the lowest MSE—was used to train a model on the full training set and tested on the test set.

The best linear regression model was also compared to a simple baseline. For difficulty, the baseline returned the mean difficulty across all surveys. For enjoyment, the baseline returned the mean enjoyment score across all surveys.

7.3 Evaluation

Out of the 150 participants recruited, 143 completed at least one level. 70 participants completed all eleven levels. On average, each player completed 7.9 levels; this average does not take into account whether the player beat the level or gave up. In the case of giving up, players gave up on 41% of the levels they played. Not considering the tutorial level, levels were played by a minimum of 1 player, a mean of 5.2 players, a median of 5 players, and a max of 10 players with a standard deviation of 2.0; a non-random level selection method would have improved this distribution.

In total, players made $3,086$ attempts to complete the levels. There were three reasons players lost: coming into contact with an enemy (71%), running out of stamina (20%), and running

[5]There are 11 heuristics, meaning $2^{11} = 2048$ possible models. A model with no inputs was not trained, resulting in 2047 total models.

Heuristic	Difficulty	Enjoyment
density	**1.0**	0.0
food-density	0.0	0.0
jaccard-no-enemies	0.0	**0.5**
jaccard-nothing	**1.0**	0.0
leniency	0.2	0.0
path-no-enemies	0.0	0.0
path-nothing	0.4	**0.8**
proximity-to-enemies	**1.0**	**1.0**
proximity-to-food	0.2	**0.4**
stamina-percent-nothing	0.2	**0.4**
stamina-percent-enemies	**0.5**	0.0

Table 7.1: Percent heuristic usage for the top-ten best performing heuristic combinations for predicting difficulty and enjoyment. The top four heuristics for both are bolded. Note the tie for *proximity-to-food* and *stamina-percent-nothing* for predicting enjoyment.

into a spike (9%). There was an overall win rate of 18%. On average, one playthrough of a level (not necessarily beating it) took 11.3 seconds. Ratings of difficulty and enjoyment can be seen in Figure 7.1. The median difficulty rating was 4, which links to "neutral." On the other hand, players gave enjoyment a median rating of 6, which links to "agree."

Agreement in level ratings was also tested. To do this, two columns were created, *agree* and *disagree*, in which each row was the sum of ratings per a level with a threshold of 4 or "neutral", where user ratings less than four went to *disagree* and user ratings greater than 4 went to *agree*; neutral ratings were not included in either category. For each level, the max of the two variables was divided by the sum, which gave *level agreement*. The mean of *level agreement* across all levels was calculated as the agreement of user ratings. Agreement on the difficulty of a level was 73.4%; players generally agreed on whether a level was difficult or not. Agreement for enjoyment was much higher at 87.2%.

7.3.1 Difficulty

The number of times a heuristic was used in the top ten heuristic combinations for a model predicting difficulty can be seen in Table 7.1. *density*, *jaccard-nothing*, and *proximity-to-*

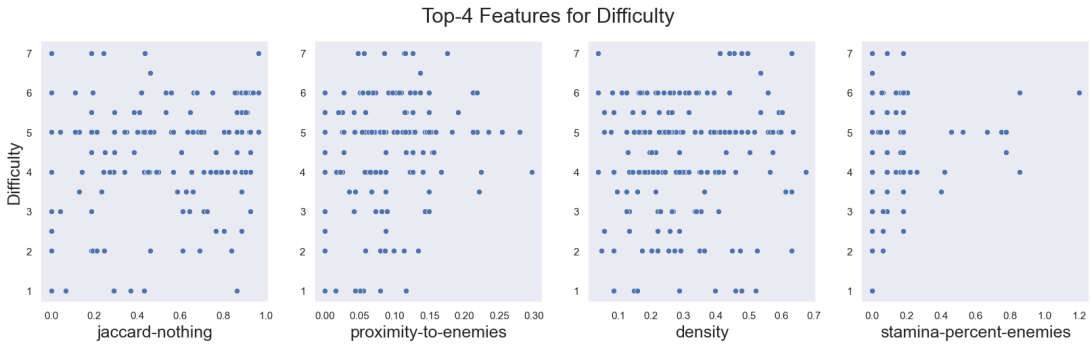


Figure 7.2: Top four features used to predict difficulty.

enemies were used by all ten. Path length was assumed to be a good representative of difficulty [59]. However, *path-no-enemies* wasn't used once, and *path-nothing* only 4 out of 10 times. Instead, using Jaccard similarity was a stronger approach; *jaccard-nothing* was used by all ten models. But, like *path-no-enemies*, *jaccard-no-enemies* was not used once by any of the top ten heuristic combinations for predicting difficulty.

Figure 7.2 shows scatter plots of the top four features—*jaccard-nothing*, *proximity-to-enemies*, *density*, and *stamina-percent-enemies*—against difficulty. *stamina-percent-enemies* was the bottom performing of the top four. When the value was equal to zero, there was a vertical line of points from 1 to 7 for difficulty. The other three did not suffer from this as much, but the phenomena was somewhat prominent for *jaccard-nothing* and *proximity-to-enemies*. There was some surprise that *density* played such a prominent role in predicting difficulty. In theory, high-density levels reduce the number of actions available to the user. If near an enemy in a high-density spot, the player has to go closer to the enemy to make it past that enemy. Further, the graph for *density* shows that while there was noise, there were no vertical lines of points, which likely improved difficulty prediction. However, each plot does not show a clear pattern and is clearly subject to noise. As a result, it was unsurprising to find that the top ten models—for difficulty prediction and enjoyment prediction—all used a combination of input heuristics rather than a single heuristic as input.

The best combination of heuristics was *jaccard-nothing*, *proximity-to-enemies*, *stamina-percent-enemies*, and *density*. Using these as input for a linear regression model, the model was tested on the test set. The median square error was 0.96. The max square error was 8.68. A value of less than one for the median is a good sign because it means that the prediction was within one category of the 7-Likert scale; the model was generally close to the user's rating of difficulty.

The baseline returns the mean difficulty from the player study. When run on the test set,

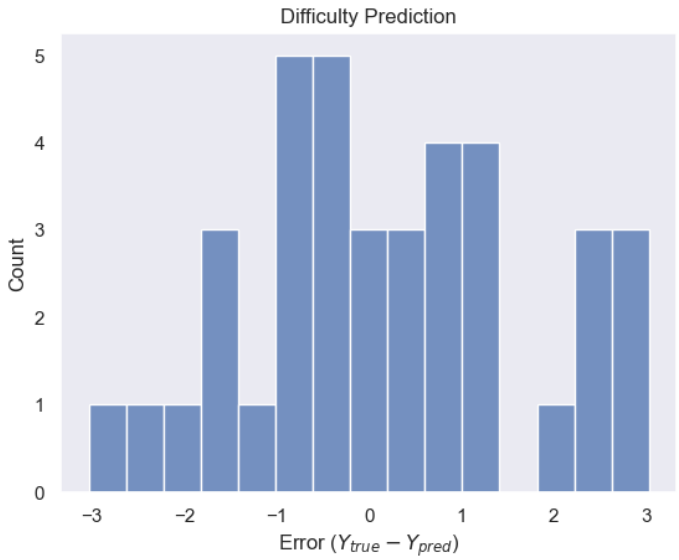


Figure 7.3: Difficulty prediction histogram for $Y_{true} - Y_{pred}$.

it had a median square error of 1.27 and a max square error of 5.80. The linear regression model had a lower median square error but a higher max square error.

The square error does not tell us whether a model was generally over or underestimating difficulty. For that, the difference ($Y_{true} - Y_{pred}$) was used. Y_{true} represents the difficulty values from the test dataset. Y_{pred} represents the predicted difficulty values using the best model. The results are in Figure 7.3. Predictions were to the left of the column associated with zero 17 times, and predictions were on the right 17 times. The error size was larger in the positive direction, meaning that difficulty was generally underestimated.

7.3.2 Enjoyment

Table 7.1 also shows the number of times a heuristic was used in the top ten heuristic combinations for predicting enjoyment. The best-performing heuristic was *proximity-to-enemies*. *proximity-to-enemies* was expected to play less of a role in predicting enjoyment and more difficulty, but it played a significant role for both. The second most used heuristic was *path-nothing*. This yields the interesting conclusion that path length was more useful in predicting enjoyment than difficulty. Also, note that *path-no-enemies* wasn't used once. There was a similar pattern where *jaccard-nothing* was a top-performing heuristic for difficulty but was not used once for predicting enjoyment. *jaccard-no-enemies* was used for five out of the ten best-performing heuristic combina-

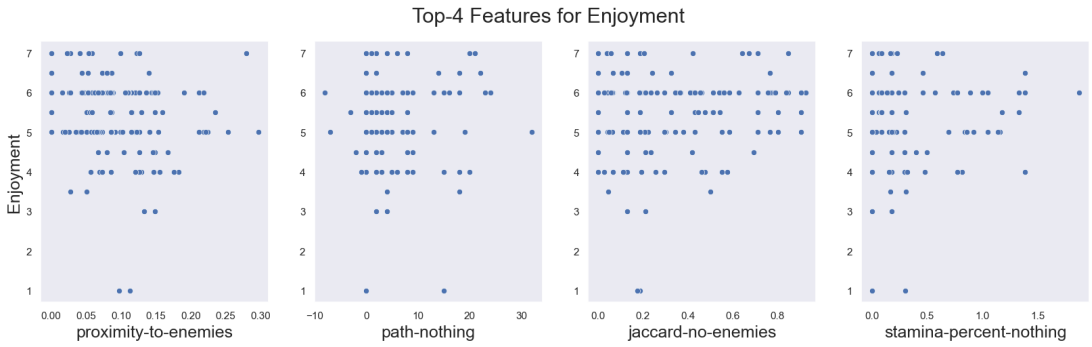


Figure 7.4: Top four features used to predict enjoyment.

tions for predicting enjoyment but not once for predicting difficulty. Only two other heuristics were used to predict enjoyment: *proximity-to-food* and *stamina-percent-nothing*. *proximity-to-food* was expected to play less of a role in predicting enjoyment, but it may be the case that acquiring extra stamina gave players a slight sense of accomplishment and/or relief while playing.

Figure 7.4 shows scatter plots for the top four features against enjoyment. In the case of all four, the vertical lines noted in Section 7.3.1 are occurring. The least pronounced of these are with *proximity-to-enemies*. *path-nothing* has more vertical lines than the other three, but it was the second-best performer. *jaccard-no-enemies* had minimal vertical lines except for when the path was unchanged. In this regard, *proximity-to-food* also behaves poorly when the path found to complete the level does not have nearby food.

The top-performing model used two heuristics: *path-nothing* and *proximity-to-enemies*. The second best-performing combination added *stamina-percent-nothing*. A linear regression model was trained with *path-nothing* and *proximity-to-enemies* as input. For the test set, the median square error was 0.79 and the max square error was 5.27. Overall, enjoyment was better predicted than difficulty, but also note that enjoyment followed a clearer distribution than difficulty, see Figure 7.1.

The baseline, which returns the mean enjoyment score from the player study, had a median square error of 0.70 and a max square error of 2.84. The baseline outperformed our best-performing model in both cases. Future work should identify better heuristics for predicting enjoyment.

Figure 7.5 shows the distribution of errors for $Y_{true} - Y_{pred}$, where Y_{pred} was calculated with a linear regression model trained with *path-nothing* and *proximity-to-enemies* as inputs and Y_{true} was the test dataset outputs. Again, this was useful because it gave information on whether models overestimated or underestimated the expected enjoyment of a level. There were 12 levels predicted to be more enjoyable than what users said and 24 levels predicted to be less enjoyable

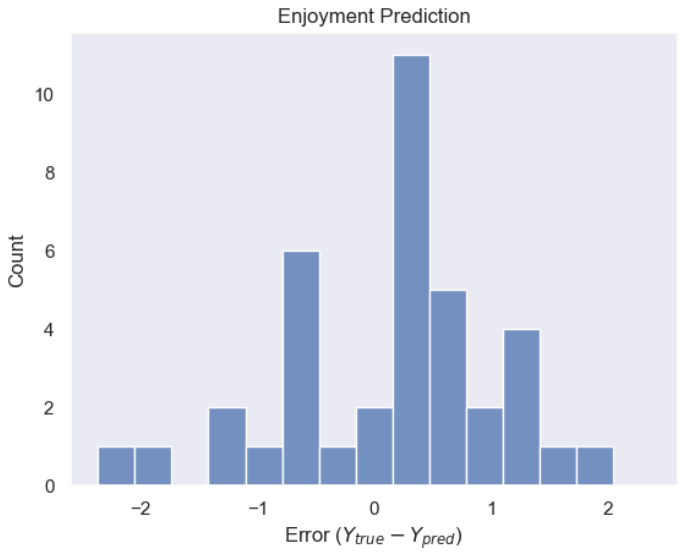


Figure 7.5: Enjoyment prediction histogram for $Y_{true} - Y_{pred}$.

than what users said. Thus, the model generally predicts that levels are less enjoyable than they are. It is also interesting to note that there are only two levels within the bin for zero error.

7.3.3 Correlation between Difficulty and Enjoyment

As noted in the related work section, past work has observed a positive correlation between difficulty and enjoyment [141], but this finding was not replicated by the work in this Chapter. Tests were run by finding the median difficulty and enjoyment for each level. The correlation between difficulty and enjoyment was tested. The Pearson correlation coefficient [29] was -0.083 with a p-value of 0.005, the Kendall rank correlation coefficient [2] was -0.122 with a p-value of 0.033, and the Spearman rank correlation coefficient [212] -0.194 with a p-value of 0.007. The coefficients were very close to zero, meaning there was little, if any, relationship between difficulty and enjoyment, and any relationship appeared to be negative. Further, each coefficient showed a low p-value, which indicates statistical significance with 190 total samples.[6] This suggests that it is incorrect to assume a positive correlation between difficulty and enjoyment of roguelikes where players must avoid enemies.

This can also be examined subjectively. Figure 7.6 shows four levels evaluated by par-

[6] As a reminder, the results from the level segment that every player played first were not used in this analysis, which is why the number is 190 and not 191.

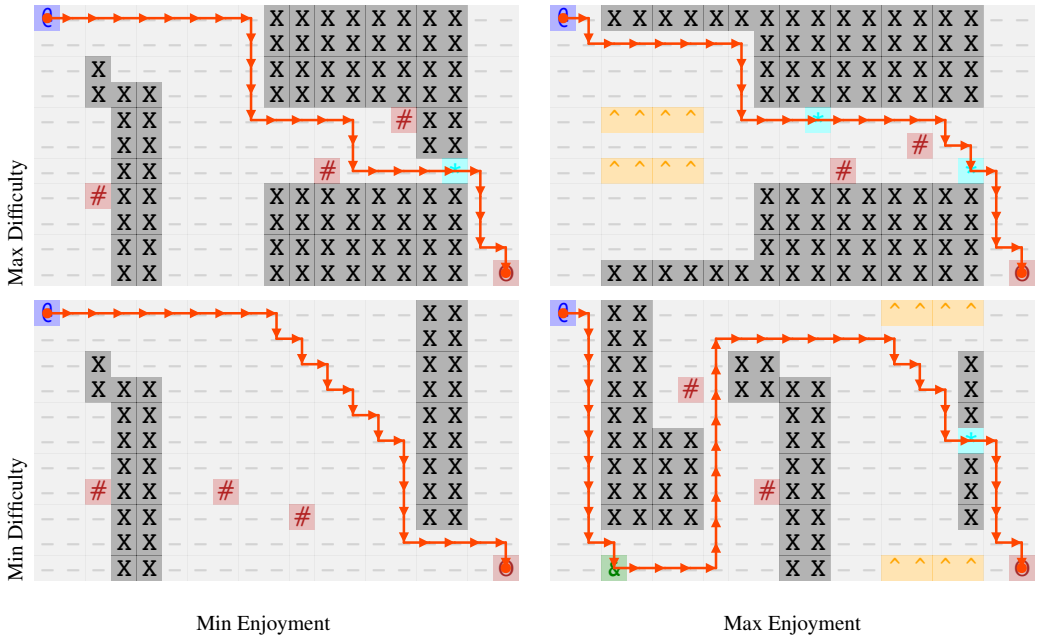


Figure 7.6: Example levels for min and max of difficulty and enjoyment based on mean ratings from study participants. Red arrows show the path found by the A* agent to complete the level.

ticipants that have been selected based on being the most or least difficult and the most or least enjoyable. Starting at the bottom left, it can be seen that the path to beat the level doesn't require any interactions with the enemies—enemies are represented by a red hashtag—to beat the level. The result is a level that requires very little thought to complete.

Going to the right to min difficulty and max enjoyment, it is clear the player is required to get food—the green '&'— to have enough stamina to beat the level, and there is also a switch—the blue '*'—which the player must come in contact with to complete the level. These additional factors contribute to the experience, but the most interesting part of the level can be seen by looking at the path. In the middle, the path passes two enemies, but notice that the agent does not move to avoid the enemies. The agent moved as if the enemies were not there and successfully completed the level.

Changing to max difficulty and minimum enjoyment, it is clear that the section with two enemies and one pathway to get to the switch was a difficult challenge. It may be that the lack of options on how to solve the movement puzzle resulted in low enjoyment and high difficulty.

Finally, the maximum of both difficulty and enjoyment featured a very similar level to the one just examined. However, it has more open space, which lends some credence to the idea that difficulty comes at the cost of fun when player options are reduced.

7.4 Limitations

- A limitation is the use of linear regression, which has an unbounded output on a problem bounded between 1 and 7. One way to address this is to convert player scores to a percentage. With the expected output bounded between 0 and 1, Beta regression [53], which can model rates and proportions, could improve prediction. Another approach is to move away from regression and instead view the problem as a classification problem with seven classes, one for each Likert response.

- The player study did not take into account the player's learning curve [112] outside of a single tutorial level. It is easily understood that the longer a player plays a game, the more they improve [180, 142], at least up to a certain point. This is partially addressed in the next chapter by adding a tutorial to *DungeonGrams*.

- While not a limitation of the work presented in this Chapter, one limitation is the idea that to effectively use *Ponos*, there would be the requirement of a player study to find player ratings of level segments. It is not always practical or feasible to run one. The next Chapter proposes an alternative approach.

- The questions asked of players to assess difficulty and enjoyment were not validated questions. It would have been better to use a survey like the Player Experience Inventory (PXI) survey [3].

7.5 Unexplored Areas of Future Work

- Rather than manually building heuristics, heuristics could be found automatically [24, 96]. We all have biases, and those biases can lead to incorrect conclusions. By building a system that can find heuristics, it may be the case that the system could find unknown heuristics that could better predict difficulty or enjoyment.

- Players had limited time with the game. A follow-up that would be interesting is to see how ratings of levels change over time, as this would show the general learning curve of *DungeonGrams*. If players could be mapped to that curve, that could be used as a kind of progression to determine which level segments the player should be playing.

7.6 Conclusion

A player study with 143 participants evaluated *DungeonGrams* levels in terms of difficulty and enjoyment on a 7-Likert scale. Most players found *DungeonGrams* enjoyable to play, but the results were more diverse for difficulty. There was not a correlation between difficulty and enjoyment for *DungeonGrams*. Nine heuristics from past work and two new ones with Jaccard similarity were used to predict difficulty and enjoyment. Heuristics that incorporated solution path information—beyond just lengths—were most useful for making predictions.

To test which heuristics were most useful, a complete ablation study was conducted where all possible combinations of feature inputs were tested with a linear regression model. The best combination of heuristics to predict difficulty was *jaccard-nothing*, *proximity-to-enemies*, *stamina-percent-enemies*, and *density*. The best combination to predict enjoyment was *path-nothing* and *proximity-to-enemies*. Of note, *proximity-to-enemies* was used by both sets of top ten models to predict difficulty and enjoyment. Further, each top-performing model predicted difficulty and enjoyment within one point on the Likert scale based on the median square error. However, the model for predicting enjoyment did not beat the baseline approach of always returning the mean enjoyment score.

As a result, the answer to **RQ2.1** found from this work is that no known heuristic can estimate the enjoyability of a level. There are, however, several approximations for difficulty that are reasonable but not foolproof. However, heuristics do not capture a player's state. For example, a player who has played *DungeonGrams* for years is going to find most levels much easier than someone who played for the first time. Further, there is always the chance that a mechanic that was hard at first became easy after enough play. As a result, heuristics used to approximate difficulty and enjoyment will always be limited. A player model, if accurate enough, is a better approach for individualized content. However, a player model approach does not work because the pipeline described in Part II is an offline approach. Therefore, the ideal approach is for players to rate level segments.

The next Chapter uses the difficulty and enjoyment ratings that were found as part of the work in this Chapter. The ratings are used as rewards for a MDP. The MDP was evaluated by players in another player study.

Chapter 8

Difficulty, Enjoyment, or Both as a Reward

> "All right, all right," I said. "Everybody is entitled to his opinion. Forgive me, all."
>
> Anthony Burgess, *A Clockwork Orange*

RQ2.2, the focus of this chapter—Is it better to optimize for difficulty, enjoyment, or both, in terms of player experience and dynamic difficulty adjustment, when assembling levels?—was the first research question that required real players, as opposed to agents, to play levels assembled by an MDP. The goal, though, was not to answer **RQ2** (Does an MDP built automatically, a handcrafted MDP with designer-set rewards, or a static progression result in a better player experience?). The goal was to analyze the effect on player experience when different reward tables were used, and the results were used to determine the reward table of the MDP for the player studies in the next chapter.

Using the ratings gathered from the previous chapter, different reward tables were built and evaluated for the *DungeonGrams* MDP. The first was the difficulty reward table, the second was the enjoyment reward table, and the third was the mean of difficulty and enjoyment rewards for each level segment. The fourth condition switched between the difficulty and enjoyment reward table every three levels played. The fifth and final condition was a baseline, and it built a policy randomly. A player study was run with these five conditions, and participants were randomly assigned to a condition at the start of their playthrough.

In addition to collecting gameplay data, participants filled out a survey, which included

the Mini Player Experience Inventory (mPXI) (see Section 8.2) to assess the player's experience while playing *DungeonGrams*. Results for both gameplay data and survey data were analyzed to see if the null hypothesis could be rejected. The majority of categories did not result in significant statistical differences between conditions, but a difference was found for a category related to the ease of control that players experienced while playing and one custom question that had players respond to the statement, "The game was too easy."

The study, after analyzing the survey data and gameplay data, found that the MDP-based approaches outperformed the *random* condition in many ways. For example, *random* resulted in the largest average for levels won per player, but the lowest average for levels played and overall average time spent playing. This was an important finding because, if *random* performed better, then the overall validity of the approach would be in question.

Comparing the MDP conditions was more difficult. The analysis to find which performed best in terms of player experience boiled down to looking for statistically significant differences from the *random* condition. The two conditions that fit this were the difficulty-reward table condition and the mean-reward table condition. Both of these conditions exhibited a higher degree of challenge for players than the other conditions. For some games, this would make using either reward table a bad choice. However, in the context of *DungeonGrams* and DDA, the goal was to keep players challenged in the name of DDA. Of the two conditions, using the mean reward table was overall better, but the differences were minor and not statistically significant.

8.1 Related Work

Work from Shu et al. [159] showed how PCG can be guided with a player experience model [208]. While they did not test with players, their work is relevant because it raises a question: Instead of rewards like difficulty or enjoyment, why not use a player experience model to build the reward table before generation? A level segment that was expected to result in a good experience would receive a high reward, and, similarly, a level expected to result in a bad experience would receive a low reward. The question is important because one goal of **RQ2.2** is to determine which condition results in the best player experience.

The main reason we didn't use a player experience model was the issue of accessibility. The hope is that someone, someday, will use the approach built for **RQ1** in a commercial game. If the final answer to **RQ2** is that developers also need to build a player experience model, then the method will likely be out of scope. Implementing a player experience model is a lot of work.

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 rightly on gameplay, performance, and bug fixes. The goal here is to find and assess simpler alternatives. However, if one day a player experience model can be pulled from a game asset store and just work, then the method will be able to work with said model with minimal effort.

One valid criticism of the approach is that the reward is not concrete. As the previous Chapter showed, difficulty and enjoyment are concepts, and numericizing 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 [206] or the likelihood that a player will succeed when playing a level segment [63]? 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* 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 [7]. 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 [11]. While the player rankings of difficulty found in Chapter 7 capture a discrete scale—i.e., Likert scale—they should still, in theory, capture the more objective aspects of difficulty. This same line of reasoning applies to enjoyment, but enjoyment is far more subjective.

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 [68]. A level may have multiple potential paths to beat it, but those paths can also be broken down into a list of required mechanics. Therefore, 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. [25], which built a system for building an automatic game progression. The system's goal is to introduce new mechanics while maintaining an appropriate level of difficulty. This is a similar, but different approach to the one built for **RQ1**. However, as the next Chapter will show, we can design the BCs of Gram-Elites to reflect challenges and obstacles of a level segment to get similar behavior.

The work from Butler et al. [25] was not a work focused on DDA; it was a work in ITS [38, 65, 122, 135]. The overlap between DDA and ITS has not, to my 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 [6], how to multiply and divide [157], how to program [26], 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 ITS was one based on the game's mechanics. 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 [25], 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 [160]. One recent solution is to use a constraint solver, which shows that a level cannot be beaten without the given mechanic [36]. 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 required mechanics, which looks at level features (i.e., gaps) and elements (i.e., a Goomba in *Mario*). However, the goal here is simplicity, but such an approach would be an interesting area for future work.

8.2 The Mini Player Experience Inventory

The PXI [3] is a validated survey designed to measure the player's experience regarding functional and psychosocial consequences. Before getting to the definitions of these two, it is helpful to understand that the PXI evaluates ten categories, five for functional and five for psychosocial.

Each construct is assessed with three questions for a total of 30 questions. Questions are responded to on a Likert Scale from strongly disagree to strongly agree.

A *functional* consequence is an effect on the player's experience that is attributable to game design choices. The PXI evaluates autonomy, curiosity, immersion, mastery, and meaning. For example, Chan et al. [27] ran a study on how high-fidelity versus low-fidelity versions of the game *Brukel*[1] can affect player experience. After playing, players were asked to fill out the PXI. Chan et al. found that high fidelity somewhat improved the player's feeling of immersion, but there was no significant change in the player's sense of autonomy, curiosity, mastery, and meaning.

Psychosocial consequences deal with second-order emotions. A second-order emotion is an emotion that comes after a primary emotion, such as joy or anger. The PXI evaluates audiovisual appeal, challenge, ease of control, goals and rules, and progress feedback.

A potential problem with the PXI is the number of questions. If there are too many questions, then participants may become fatigued and begin to answer survey questions randomly. One solution is to build a survey with fewer questions. Enter the mPXI [74], a minified version of the PXI. For each of the ten categories, the three corresponding questions were analyzed, and the best subquestion was selected for a total of ten questions:

- Autonomy - I felt free to play the game in my own way.
- Curiosity - I wanted to explore how the game evolved.
- Immersion - I was fully focused on the game.
- Mastery - I felt I was good at playing this game.
- Meaning - Playing the game was meaningful to me.
- Audiovisual appeal - I liked the look and feel of the game.
- Challenge - The game was not too easy and not too hard to play.
- Ease of control - It was easy to know how to perform actions in the game.
- Goals and rules - The goals of the game were clear to me.
- Progress feedback - The game gave clear feedback on my progress towards the goals.

In addition, Haider et al. [74] added an optional eleventh category: enjoyment. They added three questions related to enjoyment and selected: "I had a good time playing this game."

[1] https://store.steampowered.com/app/1073900/Brukel/

Figure 8.1: Screenshot from the tutorial added to *DungeonGrams*. Note also the red area around the enemy. This space indicates the detection radius of the enemy (#) and changes color to a bright red—as seen in this screenshot—when the enemy can move the next turn.

8.3 Updates to *DungeonGrams*

Several updates were made to *DungeonGrams* from when the study in Chapter 7 was run. The updated version of the game is playable online.[2]

1. Screen Size - The game's resolution was made wider to display two level segments. Before, the screen was only wide enough to display one level segment. A scrolling camera was not an option because *DungeonGrams* requires the player to plan their route through the level. Incomplete knowledge could result in players failing levels they would have otherwise beaten. It was hypothesized that this would have resulted in a frustrating experience for the player. Further, making the screen wider was easier than implementing a camera.

2. Tutorial - A lesson learned from the previous Chapter was that there should have been a tutorial built into the game—something to make the onboarding process smoother for players. As a result, a tutorial was added. Figure 8.1 shows the final step of the tutorial. Before being asked to reach the portal, the player is taught to move with prompts and to hit a switch to open the portal. It is left to the player to discover food (&) and learn that every switch must be reached to open the portal.

3. Enemy Territory - A frustrating aspect of playing *DungeonGrams* was that players had to use the turn number (top right of Figure 8.1) to figure out if the enemy's turn was next. Further,

[2] https://bi3mer.github.io/dg/

without checking the code, there was no way for a player to know when an enemy would go after them versus when they would not. This was addressed by adding a territory marker around the enemy. When the player is in range of the enemy, the territory shows up on the screen. If the enemy does not move in the next turn, the red territory is visible but blends in with the background. If the enemy will move the following turn, the terrority is a brighter red, as seen in Figure 8.1.

4. End State - *DungeonGrams* was updated to have an *end* state. Reaching the *end* state means beating the game. The *end* state was found automatically by finding the state with the largest sum of mean difficulty and mean enjoyment from the player study run in the prior Chapter. A *player-won-state* was added to the game in case any player in the study reached the end state.[3]

8.4 The Player Study

A player study was run with a total of 250 participants using methods approved by Northeastern's Institutional Review Board. Participants were recruited via Prolific with a payment of two dollars.[4] Participants started by going to a webpage where they were asked to agree to a consent form. Afterward, they were taken to the game's webpage with a random UUID. The participant could quit anytime by pressing a button labeled "Go to survey." Alternatively, if the player played for more than ten minutes, they would also be taken to the survey.

Participants first played through the tutorial described in Section 8.3. The tutorial was not skippable unless the player quit with the "Go to Survey" button. After completing the tutorial, the game started. Note that the time it took the player to complete the tutorial was considered part of the player's overall playtime; if the player spent a minute playing through the tutorial, then they would have nine minutes left to play the game.

After the tutorial, players were randomly assigned one of five conditions described in Section 8.4.2. Each condition mapped to a director. The director generated levels for the player using an MDP, described in section 8.4.3. Unlike in the previous version of *DungeonGrams*, the game had no functionality for the player to give up on a level or restart. Players had to fail or win,

[3]No player in the player study beat the game, but it was possible. During testing, I was able to complete the game in under five minutes.

[4]https://www.prolific.com/

and afterward, the game announced the result. It was in this announcement state that playtime was assessed. If the maximum time of ten minutes was reached, the player would be taken to the survey.

The survey was composed of four pages:

1. The first page covered demographics. Each question was selected from a set of pre-approved questions.
   (a) "What is your age range?" And participants could respond with one of the following: "18-24", "25-35", ..., "95 and above", or "Prefer not to answer."
   (b) "Approximately how often per day, on average, do you spend playing computer games?" And participants could respond one of the following: "Less than 10 minutes", "Less than 30 minutes", "Less than 1 hour", "Less than 2 hours", "Less than 3 hours", "3 or more hours", or "Prefer not to answer."
   (c) "What kinds of games do you play? Check all that apply:" And the players could check any of the following: "Action/Adventure", "Casual", "Citizen Science", "Puzzle", "Role-playing", "Shooter", "Other", "None of the above", and "Prefer not to answer." The latter two options were exclusionary, and behavior was added so that all the other boxes were unchecked if either was selected. If any of the other boxes were checked, then both were unchecked.
2. The second page had three required statements for the user to respond to on a 7-point Likert scale from "Strongly disagree" to "Strongly agree."
   (a) "The game was too hard."
   (b) "The game was too easy."
   (c) "I felt bored while playing this game."
3. The third page consisted of all eleven questions from the mPXI survey.
4. The final page displayed a Prolific code with a link that the user could click to receive their payment.

The second page of the survey consisted of three questions. The first two questions are directly related to the Challenge survey item: "The game was not too easy and not too hard to play." The two questions addressed one potential problem: if we obtained a significant result for Challenge,

it would not be possible to say "X was more challenging than Y" or "X was less challenging than Y." The last question dealt with boredom. Its addition was slightly redundant, as boredom is likely inversely related to curiosity. However, a direct measure of boredom was warranted because this approach alters the typical video game loop. The typical loop in a game like *DungeonGrams* is a try/fail cycle. Players play a level, lose, and keep playing until they eventually win. This cycle can be frustrating, but it can also be highly engaging as players overcome challenges. The system in this dissertation does not eliminate this cycle, but it can be more discrete. It was, therefore, worth considering whether the overall result of playing with an MDP-based condition was a more boring experience for the player.

Each participant was paid $2.00. No identifying information was requested or stored.

8.4.1 Gameplay Data

In addition to survey data, gameplay data was tracked and stored. A log was sent to a server after every level played and contained the following:

- *playerID* - the UUID assigned to the player after signing the consent form.
- *diedFrom* - if the player died, then this would have a string to identify how they died (e.g., "stamina" would mean the player died from running out of stamina).
- *staminaLeft* - how much stamina the player had at the end of the session.
- *time* - how many seconds the player played the level.
- *won* - a boolean for whether the player won or not.[5]
- *order* - the order in which the player played the level associated with this log entry. The order is incremented from 0.
- *levels* - a list of the level segment ids, in order, that make up the level.
- *pathX* - a list of the x coordinates that the player took while playing the level.
- *pathY* - a list of the y coordinates that the player took while playing the level.

[5]This is redundant when considering the *diedFrom* field.

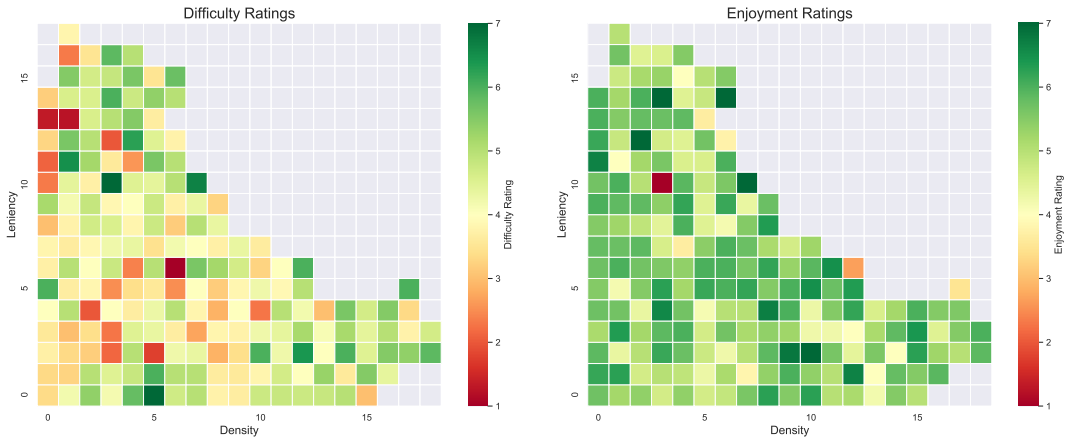


Figure 8.2: Two heatmaps to show the rewards for when the Markov decision process uses difficulty (left) and enjoyment (right) as a reward. The values are the average user rating of the given level segment.

8.4.2 Study Conditions

There were five conditions that a participant could be assigned to:

- *difficulty* - For every state, the reward was the mean of the player-rated difficulty from the player study in the previous chapter.[6]

- *enjoyment* - For every state, the reward was the mean of the player-rated enjoyment.

- *mean* - For every state, the reward was the mean of the player-rated difficulty and enjoyment.

- *switch* - The rewards of the MDP swap between *difficulty* and *enjoyment* every three levels played.

- *random* - This is the random director, and it creates a random policy before every level generation task.[7] However, unlike in the study with agents in Chapter 5, the *random* director was given the benefit of adapting to player failure by removing player edges for a fairer comparison.

8.4.3 Markov Decision Process

The MDP was formed from the level segments rated in the previous Chapter by participants. Chapter 7 evaluated 191 levels, but we did not use all these levels in the player study. For the player study, the MDP states were filtered such that every state could be reached from the *start* state and every state could reach the *end* state. The *end* state was determined automatically by selecting from the filtered states the state with the largest sum of *difficulty* and *enjoyment*.

There were 17 states with no outgoing edges in the digraph after linking. These were removed from the MDP. The reduction resulted in a total of 174 level segment states. Additionally, there were three other required states: (1) the start state, (2) the death state, and (3) the end state (i.e., the player beat the game). In total, the MDP had 176 states.

On average, every state had 3.159 outgoing edges. This average was higher than one might expect if comparing to the *Icarus* MDP in Chapter 5, which had an average of 1.877 outgoing edges. The reason for the higher count was the removal of linking states.[8] Instead, links were added to edges. This change simplified the implementation but created a problem: What if the player failed during a link? For example, the player played a level that was the concatenation of A, $link_{A,B}$, and B, and they lost during $link_{A,B}$. This was resolved by considering the link part of state A. If the player failed during the link, the implementation said that the player failed level segment A. Formally, this is incorrect. However, this decision was made based on the informed assumption that links were small and easy to complete, and if the player failed, it was likely because they made a mistake in level segment A.

The rewards for the MDP were difficulty or enjoyment ratings from the previous Chapter. Figure 8.2 shows a visualization of both. Both versions did not follow a neat gradation between the ratings like what was built in Chapter 5. Instead, the distribution of ratings was messier. Difficulty had more variability than enjoyment.

The rewards shown in Figure 8.2 were not used in the final MDP. Instead, values where transformed: $r_{new} \leftarrow (r_{old}/7) - 1$. The values were normalized to be between zero and one and then subtracted by one to be negative or zero. A level segment with a reward of 1 became -0.857. A level segment with a reward of 7 became 0. The decision to test and use this transform came from experimental playthroughs. Positive rewards led to gameplay sessions that felt unchallenging. The

[6] Using difficulty is effectively a DDA system.

[7] As discussed in Section 5.6, since the MDP is two-dimensional, minimal exploration is expected due to the phenomena of a random walk agent continuously returning to its origin in a two-dimensional space.

[8] Linking states can only have one outgoing edge.

change to a negative reward resulted in a better experience, in my opinion.[9]

The reward for the *death* state was set to -5 and the reward for the *end* state was 0. Both of these were found experimentally.

8.4.4 Updates to Policy Generation

- Typically, a policy has the structure of states linked to a single action. In Chapter 5, this became a level segment and another level segment. This was modified so that the output policy π had a level segment linked to an array. The expected utility calculation was still done, but if two or more states had the same expected utility, then they were all placed into the array as potential outputs. Then, when using this policy, a level segment was selected randomly from the array. This added some variability to the gameplay while still being optimal according to the MDP.

- The first implementation of API removed edges from the *start* node to level states based on the largest reward when the player failed. However, when playtesting, this behavior resulted in undesirable results given a reward space that didn't follow a nice gradient, as shown in Figure 8.2. A new field called *depth* was added to every state, which was used for edge removal instead. Depth was calculated with a BFS from the start node to every target node.

- The initial behavior for API state removal was to remove edges to states on first failure. This behavior was removed due to being highly frustrating when playtesting. It was replaced with a one-level buffer, so edges could only be removed after the player had failed at least twice, and the number of edges to be removed was reduced by 1, so if the player had failed twice in a row, only one edge was removed instead of two.

- Another problem that quickly became apparent when playtesting was the constant replaying of previous level segments that had already been beaten. This occurred because the first level segment selected by the policy was from the *start* node, which can only be connected to level segments that the player has beaten.[10] After beating a level, the next level would always start with a level segment that the player had just beaten. This was resolved by adding a

[9] This is analogous to the example given by Russell and Norvig in their textbook [145]. They showed that positive rewards led to a policy that did not go toward the end because the reward was infinite, given an infinite time horizon. This resulted in a risk-avoidant policy. Negative rewards, depending on the degree relative to the final reward, resulted in a policy being more or less risk-taking.

[10] The exception is at the start of play when the player has not beaten any levels. Or, the player has failed so much that the *start* node only has one outgoing edge.

check to see if the player had just won, and if so, the first level segment selected was found with `choice(pi[choice(pi[START])])`[11] instead of `choice(pi[START])`, essentially skipping the *start* state when building the next level for the player to play.

$$P(s_{tgt}|s_{src}, s_{tgt}) \leftarrow \frac{1 + \sum_{\{s,s_{tgt}\}\in E} C(s, s_{tgt})}{1 + \sum_{\{s,s_{tgt}\}\in E} \textit{visits}(s, s_{tgt})} \tag{8.1}$$

- The final update fixed a general weakness in the way that the probability of success and failure was calculated in edge transitions. Equation 8.1 updates the calculation—the original equation is Equation 5.2—by using the percent complete calculation instead of a binary for whether the player won or lost. This allows for a more fine-grained calculation of the likelihood of player failure or success, as a player who lost early on is less likely to win than one who made it much further in the level segment.

8.5 Results

The study was run on Prolific with batches of twenty-five players over an hour. Batches were done in case any bugs or errors were discovered either through the game's logs or reports from the players. No bugs were found. There was one report from a participant, though. They could not click "I Agree" on the consent form because they used an incompatible browser. This participant dropped out of a study, and a replacement was added.

On average, players participated in the study for a total median time of 340 seconds. This resulted in an average pay of $21.18 an hour. The time spent playing the tutorial and filling in the survey affected the total time spent playing the game, and the time participants spent playing the game is analyzed in Section 8.5.3.

A player could reach the *end* state and beat the game in under ten minutes. It was, therefore, possible that a participant could beat the game. However, no player in the study accomplished this.

8.5.1 Demographics

The sample of participants's ages consisted of 21 (12.57%) between 18-24, 62 (37.13%) between 25-34, 40 (23.95%) between 35-44, 27 (16.17%) between 45-54, 15 (8.98%) between 55-

[11] `pi` represents a policy.

Condition	Participants	Filtered Count
random	64	41
difficulty	54	31
enjoyment	49	38
mean	49	33
switch	36	24

Table 8.1: Participant counts for each condition before and after filtering for players that completed at least one level, after finishing the tutorial.

64, 2 (1.20%) between 65-74, 0 between 75-84, 0 between 85-94, and 0 between 95 or above. No participants answered "Prefer not to answer" for their age range.

In response to "Approximately how often per day, on average, do you spend playing computer games?", 23 (13.77%) participants responded "Less than 10 minutes," 27 (16.17%) responded "Less than 30 minutes," 34 (20.36%) responded "Less than 1 hour," 42 (25.15%) responded "Less than 2 hours," 18 (10.78%) responded "Less than 3 hours," 21 (12.57%) responded "3 or more hours," and 2 (1.20%) responded "Prefer not to answer."

8.5.2 Likert Data

Before the analysis was run for Likert data, players were filtered out if they did not complete at least one level. Table 8.1 shows the original number of players for each condition and the amount after filtering. Note that the player must have completed the tutorial to have beaten a level. The table shows an unfortunate distribution where *switch* received 36 participants and *random* 64. The code for condition selection was tested and reviewed, but we did not find any problems with the code.[12]

Analysis of all Likert data was performed with the Kruskal-Wallis test [119]. This was done because ANOVA [169] assumes continuous 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 [48] with a correction via the Holm method [80] to find any statistically significant differences between groups.

The majority of survey questions from the mPXI had a p-value of ≥ 0.05: Audiovisual Appeal, Challenge, Clarity of Goals, Progress Feedback, Enjoyment, Autonomy, Curiosity, Immer-

[12] return list[Math.floor(Math.random() * list.length)]; Note that the range of Math.random is $[0, 1)$.

	random	difficulty	enjoyment	mean	switch
Audiovisual Appeal (p=0.944)	**0.0** (2.0)	**0.0** (1.0)	**0.0** (1.0)	**0.0** (2.0)	**0.0** (1.5)
Challenge (p=0.659)	**2.0** (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)
Ease of Control (p=0.043)	**3.0** (0.0)[a]	2.0 (1.0)[a]	2.0 (1.0)[a]	2.0 (1.0)[a]	2.0 (1.0)[a]
Clarity of Goals (p=0.388)	**3.0** (0.0)	**3.0** (0.0)	**3.0** (0.0)	2.0 (1.0)	2.5 (0.5)
Progress Feedback (p=0.413)	1.0 (2.0)	1.0 (1.0)	**2.0** (1.0)	1.0 (1.0)	1.0 (1.0)
Enjoyment (p=0.347)	**2.0** (1.0)	1.0 (1.0)	1.5 (0.5)	**2.0** (0.0)	1.0 (1.0)
Autonomy (p=0.738)	**2.0** (1.0)	1.0 (1.0)	1.0 (1.0)	1.0 (1.0)	1.5 (1.0)
Curiousity (p=0.749)	**1.0** (1.0)	**1.0** (1.0)	**1.0** (1.5)	**1.0** (1.0)	**1.0** (1.0)
Immersion (p=0.176)	**3.0** (0.0)	**3.0** (0.0)	2.5 (0.5)	2.0 (1.0)	2.0 (1.0)
Mastery (p=0.181)	**1.0** (1.0)	-1.0 (1.0)	**1.0** (1.0)	0.0 (1.0)	0.5 (1.5)
Meaning (p=0.521)	**0.0** (2.0)	**0.0** (1.0)	**0.0** (1.5)	**0.0** (2.0)	-1.0 (1.0)
Bored (p=0.659)	-1.0 (2.0)	-1.0 (1.0)	-1.0 (1.5)	-2.0 (1.0)	**-0.5** (1.5)
Difficult (p=0.122)	-2.0 (1.0)	-1.0 (1.0)	-1.5 (0.5)	-1.0 (1.0)	**0.0** (1.5)
Easy (p=0.007)	**0.0** (1.0)[a]	-1.0 (1.0)[b]	-1.0 (1.0)[ab]	-1.0 (1.0)[b]	-1.0 (1.0)[ab]

Table 8.2: Results for all survey questions related to Likert data for participants who completed at least one level. The format for each condition is "[[median]] ([[median absolute deviation]])". The largest median for each survey question is bolded. If there was a p-value < 0.05, then groupings are displayed with a compact letter display.

	Win Path Length
random	65.067 (14.917)[ab]
difficulty	62.562 (11.616)[ab]
enjoyment	67.368 (22.092)[ab]
mean	61.203 (10.906)[a]
switch	**69.150** (17.591)[b]

	Levels Played
random	7.898 (6.777)
difficulty	8.286 (5.771)
enjoyment	8.791 (5.975)
mean	**9.650** (6.858)
switch	7.500 (7.070)

	Levels Won
random	**5.073** (4.640)
difficulty	3.613 (2.043)
enjoyment	4.868 (3.122)
mean	3.471 (2.648)
switch	3.333 (3.412)

	Deaths
random	3.891 (3.225)[a]
difficulty	5.756 (3.681)[b]
enjoyment	4.707 (3.133)[ab]
mean	**6.872** (4.794)[b]
switch	5.370 (4.038)[ab]

	Time Played per Player
random	134.238 (109.367)
difficulty	142.136 (103.704)
enjoyment	163.762 (125.794)
mean	**188.694** (151.888)
switch	156.965 (139.967)

	Time Played per Level
random	16.997 (12.516)[a]
difficulty	17.154 (10.667)[ab]
enjoyment	18.629 (11.142)[b]
mean	19.554 (15.688)[ab]
switch	**20.929** (27.789)[ab]

Table 8.3: Quantitative results from the player study. Results are in the format of "[[mean]] ([[standard deviation]])". Each category failed to pass the Shapiro-Wilk test and/or Levene's test; therefore, the Kruskal-Wallis test was used for every category. A compact letter display [66] shows similar groups based on a posthoc analysis with Dunn's test if the p-value from the Kruskal-Wallis test was < 0.05.

sion, Mastery, and Meaning. Of the three custom questions added, the questions related to boredom and difficulty both had a p-value ≥ 0.05 as well.

Easy - As a reminder, this survey question asked participants to reply to: "The game was too easy." The post-hoc test gave two groups, and *enjoyment* and *switch* were in both. *random* was significantly different statistically to *difficulty* and *mean*, with *random* being rated as more easy to play. It is worth noting that both Challenge—from the mPXI survey—and the custom difficulty survey question had p-values ≥ 0.05, while the easy variant did not. Why this occurred is unclear, but it may be easier for someone to admit that something was not too easy instead of too hard.

Ease of Control - The p-value was < 0.05, but the post-hoc test found no dissimilar groups, and it is, therefore, difficult to come to any strong conclusions.

8.5.3 Quantatative Data

No players were filtered out from this section. However, a player who did not complete the tutorial would have no data logged for playing the game.

Figure 8.3 shows tables for all the quantitative data. All the variables here are numerical. As a result, data was analyzed first to see if ANOVA [169] could be run. This was tested by running

Levene's test [57] to see if the distribution was homoscedastic and the Shapiro-Wilk test [155] to see if the data was approximately normally distributed. In all cases, the variables examined resulted in at least one of the two tests failing. The Kruskal-Wallis test [119] was run in place of ANOVA. For the variables that had a p-value < 0.05—which were "Win Path Length", "Deaths", and "Time Played per Level"—a post-hoc analysis was run with Dunn's test [48] with a correction via the Holm method [80] to find any statistically significant differences between groups.

Win Path Length - The Shapiro-Wilk test failed for this variable. The Kruskal-Wallis test p-value was 0.015, and Dunn's test was performed. A statistical difference was only observed between *mean* and *switch*, with *switch* having a longer mean path when the player won.

Levels Played - Levene's test and the Shapiro-Wilk test failed for this variable. The Kruskal-Wallis test p-value was 0.445. Meaning the null hypothesis could not be rejected.

Levels Won - Levene's test and the Shapiro-Wilk test failed for this variable. The Kruskal-Wallis test p-value was 0.079.

Deaths - Levene's test failed for this variable. The Kruskal-Wallis test p-value was 0.005. Group *a* was associated with a lower death count and group *b* with a higher death count. *random* was in group *a*, which also included *enjoyment* and *switch*. *difficulty* and *mean* were only in group *b*. *enjoyment* and *switch* were in both groups.

Time Played per Player - Levene's test and the Shapiro-Wilk test failed for this variable. The Kruskal-Wallis test p-value was 0.629.

Time Played per Level - The Shapiro-Wilk test failed for this variable. The Kruskal-Wallis test p-value was 0.010. There were, again, two groups. *random* and *enjoyment* were statistically different from each other, with *random* resulting in less time played per a level. This result, referring to players spending less time to play a level assembled in the *random* condition, was unsurprising because it was much more likely for a player to replay a level segment—see Section 8.5.4—and, therefore, know exactly how to beat it.

Condition	Total Unique Level Segments	Mean per Player	Median per Player
random	22	5.490 (2.977)	5.0
difficulty	22	7.595 (5.062)	7.0
enjoyment	**41**	**11.465** (7.092)	**11.0**
mean	38	8.725 (6.172)	6.0
switch	26	7.300 (5.599)	5.0

Table 8.4: Shows the number of unique level segments seen per condition, and the number of unique level segments that participants saw while playing per condition in terms of mean and median. The mean column includes the standard deviation in parentheses.

8.5.4 Exploration

No players were filtered out from this section. However, a player who did not complete the tutorial would have no data logged for playing the game.

Table 8.4 shows the number of unique level segments visited by participants for each condition, as well as the mean and median number of unique level segments visited by each player for each condition. *enjoyment* was the best performer for all three. *mean* was fairly close regarding the number of unique level segments seen by the player, but the mean unique levels seen per player was 3 fewer level segments, and the median was 5 less. *difficulty* and *random* were the worst performing. Neither result was surprising. *random* was expected to perform poorly based on the results in Chapter 5. *difficulty* was expected to cause more player failure, resulting in less exploration unless the player played more levels. Since players played on average 8.286 levels total in the *difficulty* condition, the result of a low number of unique level segments seen is unsurprising.

Figure 8.3 shows heatmaps of level segments reached by participants for each condition. The first thing to note is that the explored area is smaller than what we saw in Chapter 5. There were two attributable reasons. First, participants were playing the game for the first time and had a limited time to figure out the mechanics required to beat the more difficult levels. Second, players played an average of 9.613 levels across all conditions, whereas the study with agents guaranteed 50 levels played.

The counts for the level at the origin $(0, 0)$, which is to say the bottom-left, are, in part, representative of the number of participants that each condition received. The low count for *switch* is an example of this. Where things are more interesting is the origin for *enjoyment* and *mean*. Both had 49 participants, but enjoyment was filtered to 38 and mean to 33. *mean*, despite having fewer

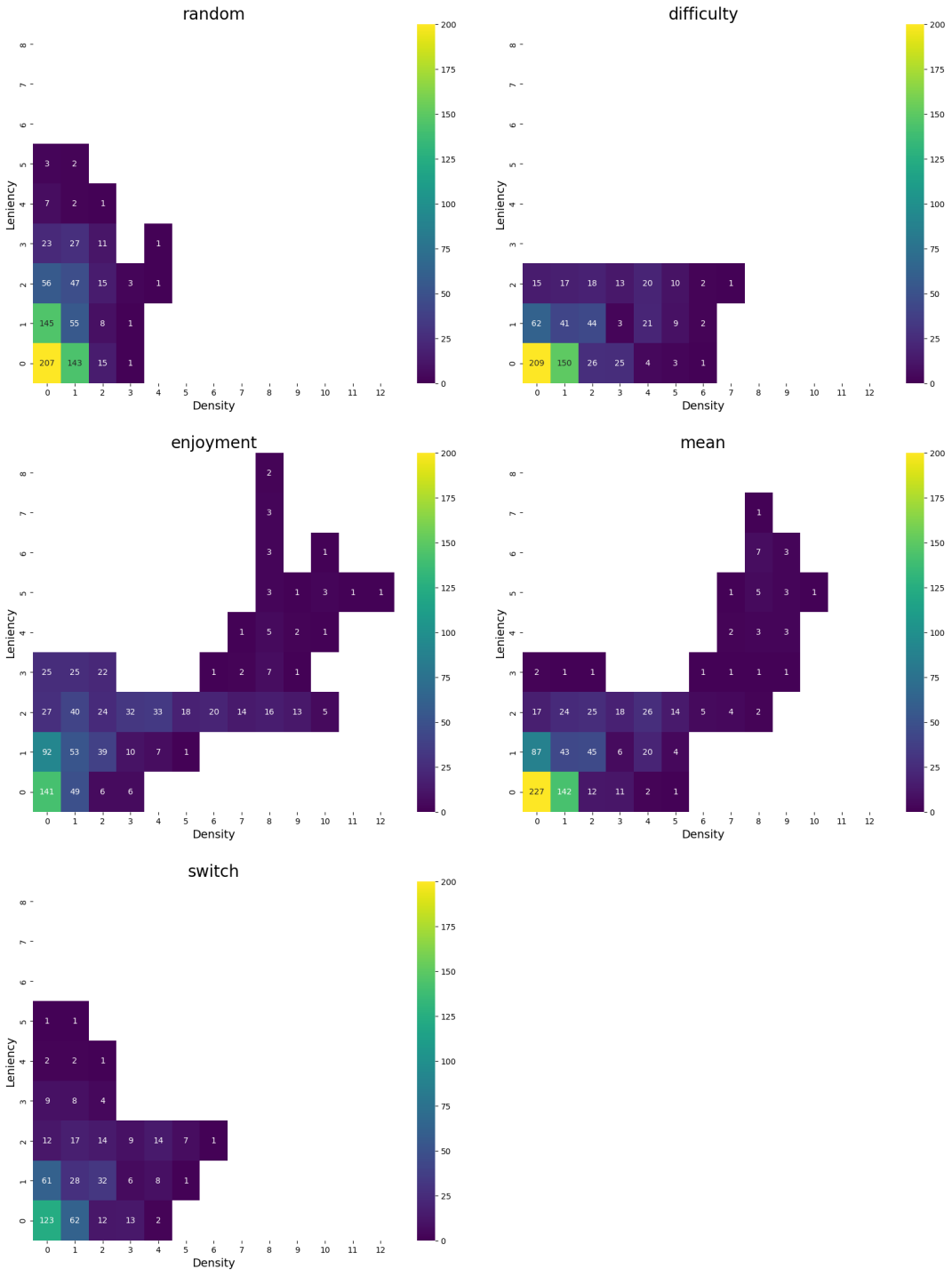


Figure 8.3: Heatmaps for each condition, which shows the number of times a given level segment in the MDP was played.

players, had the highest number of repeats at the origin with 227, whereas *enjoyment* only had 141. This is because of the choice of going up to $(0, 1)$ or the right $(1, 0)$. The level segment at $(0, 1)$ is relatively empty, but there is a switch that the player has to hit to open the portal. The level segment at $(1, 0)$ has a tricky puzzle for newcomers where the player must maneuver around an enemy to hit a switch before going to the portal. Going up, therefore, is more beginner-friendly and more likely to result in player success, whereas going to the right is the opposite.[13] This is relevant because *enjoyment* always started by going up and *mean* by going to the right. *mean* goes to the right because the level segment at $(1, 0)$ has a higher difficulty score than enjoyment (see Figure 8.2). The result is more player failure for *mean*.

This likely also explains why only 57% of participants beat a level in the *difficult* condition. Players who failed were shown the same challenging level too many times and then gave up. The *mean* condition helped address this by taking in a different reward, which allowed the MDP to find a policy that better adapted to the failure. It also indicates that the level segment at $(1, 0)$ was too difficult. If this were a game that was meant to be released, the next step would be to adjust the behavioral characteristics and re-run or manually tweak the rewards.

Looking up and to the right of the heatmaps for *enjoyment* and *mean*, we can see that both performed well because each had a few participants who excelled. A drop-off occurs around $(6, 2)$ for *mean*. The same applies to *enjoyment* when leniency is > 2. Figure 8.4 displays heatmaps for each condition, and each heatmap displays the number of unique players that played a level segment. These confirm that there were only a few players who truly explored deeper into the space. Therefore, it can be considered a "luck of the draw" that these specific players played for a *mean* and *enjoyment*. However, even without these "elite" players, it can be seen from the unique player counts that *mean* and *enjoyment* resulted in overall more exploration.

8.6 Discussion

One challenge for any DDA system is the speed with which it has to adapt to a player. The ideal system would be able to adapt to a player after seconds of gameplay. The system presented

[13] This is an example of one of the stickier elements of the approach. Ideally, it shouldn't be such a big deal if the first level segment has higher leniency or higher density, but it is given the set of BCs described back in Chapter 3. The solution is for designers to iterate by building and refining BCs while playtesting both by themselves and with outside players. Further, there is also an inconsistency. One would expect an increase in leniency to result in a higher difficulty level than an increase in density. However, difficulty for *DungeonGrams* has less to do with how many enemies or switches there are and more to do with how constrained the player's movement is. As a result, there are cases where high density levels are more difficult than high leniency.

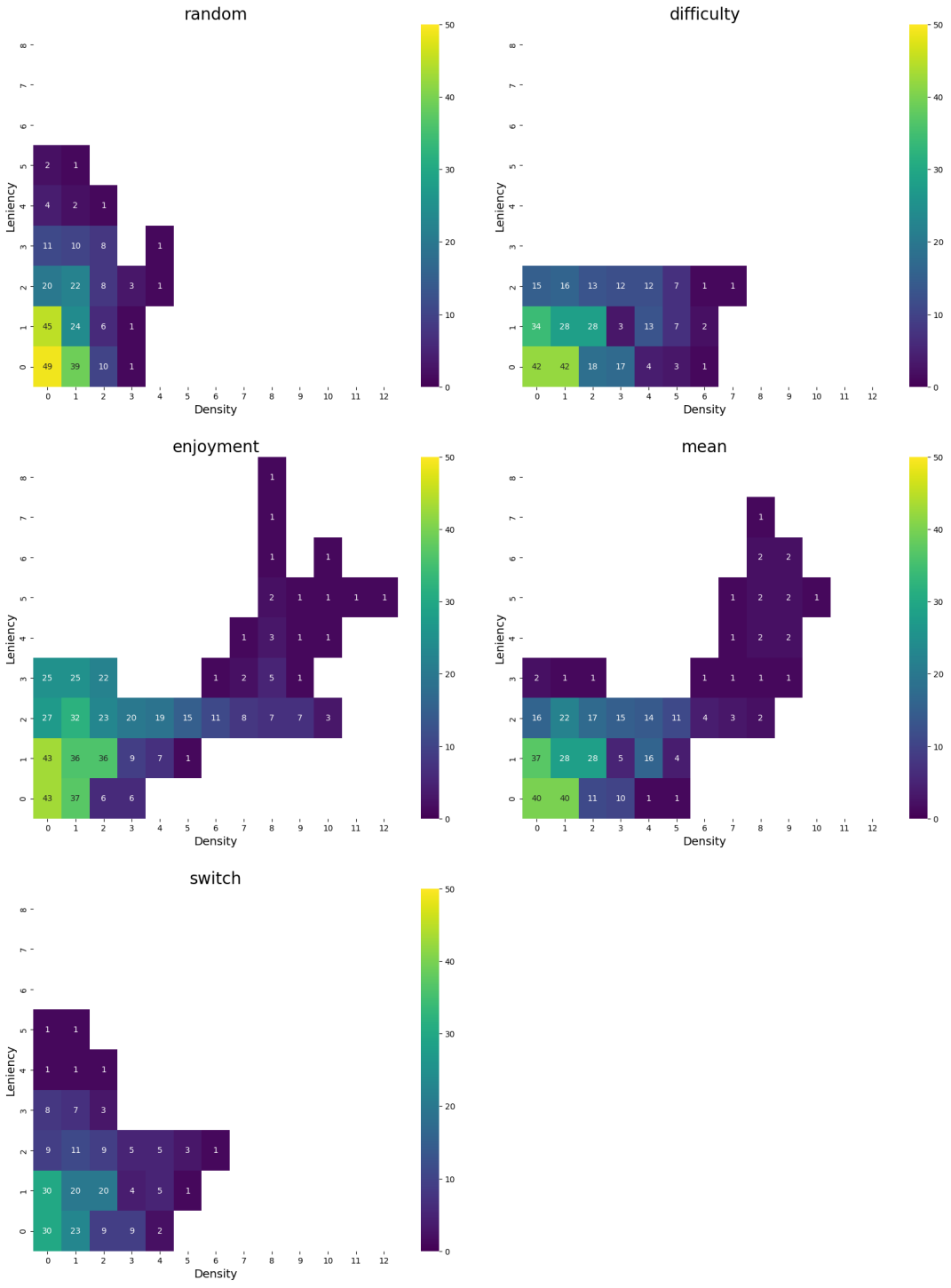


Figure 8.4: Heatmaps for each condition, which shows the number of unique players that reached a given level segment in the MDP.

can not, by design, adapt so quickly. It learns about the player and adapts to them one level segment at a time. The more the player plays, the better the MDP adapts. However, the contents of the MDP determine how successful adaptation will be. For example, the level at $(0, 1)$ was too difficult for the majority of new players. Ideally, such a difficult level would be further along in the level progression to give players time to learn the mechanics.

This abrupt progression of difficulty may, in part, be related to why participants did not play a lot of levels, as shown by the time played and number of levels played per participant in Figure 8.3. In addition to updating the MDP, changes to the study's design can be made to improve playtime and levels played. Pay can be increased, and the expected play time in the instructions can be as well. A minimum time can also be enforced [89]. A tiered payment system [133] could be added where participants are paid more based on the number of levels they played. (This could be gamed by participants, though, so it may be better to say the number of levels that the player won.)

However, despite the low number of levels played, we were fortunate enough to find significant statistical differences between conditions. Except for "Win Path Length," the groups found to be different were between *random* and one or two of the MDP-based conditions. Therefore, we can start by assessing whether *random* or the MDP approach was better based on the overall goals of DDA: the player should be challenged by the levels, but not so much that they become disengaged.

Starting with the Likert data, the only post-hoc test that resulted in multiple groups was the custom question, which asked if the game was too easy. *mean* and *difficulty* were the only conditions that were statistically different from *random*, with *random* having a median score of 0.0 and the other two a median score of -1.0. The same grouping occurred when testing for the number of deaths per participant, with *random* having 3.891, *difficulty* 5.761, and *mean* 6.872—*random* resulted in fewer deaths, and the other two resulted in more player deaths. This lines up nicely with the results for the question related to the game being too easy, since we expect players who die more to report having a harder time with the game they are playing. The three had no significant statistical difference regarding time played per level. Based on these facts, the decision is pretty clear. Both *difficulty* and *mean* challenged their players, but not to such an extent that the challenge was overwhelming. However, which of the two was better?

Except for "Solution Path Length," *difficulty* and *mean* were placed into the same group for every post-hoc test that was run. Therefore, more data would be required to say, definitively, whether one or the other was better for *DungeonGrams*. However, even though some differences were not statistically significant, it does not mean that the differences cannot be analyzed subjec-

tively. The Enjoyment category resulted in *difficulty* with a median of 1.0 while *mean* had a median of 2.0. In contrast, *difficulty* had a median of 3.0 for Immersion, while *mean* had a median of 2.0. However, for Mastery and Bored—a low score for Boredom is preferable since the goal is for the player not to feel bored—*mean* outperformed *difficulty*. Therefore, overall, *mean* performed better than *difficulty*.

This, though, is ultimately a value judgment for the designer to make and test until they are satisfied. If the designer values Immersion, then they should try *difficulty*. If they think none of the rewards tested are right for their game, then they should try a different reward. For example, another way to design the reward function is based on exploration. Then *enjoyment* or *mean* would be the best choices. However, a custom reward based on exploration could also be added, which will be done in the next Chapter.

8.7 Conclusion

Overall, the MDP-based approaches worked, but there were problems. First, the overall structure of the MDP, based on the BCs, resulted in a progression that was not well-suited to teaching players how to play the game. Second, the conditions received an unequal number of participants based on a random selection, which resulted in an uneven distribution. Third, the study's design limited play duration, resulting in a borderline insufficient number of levels played. All of these problems can be resolved in future work.

1. The MDP can be rebuilt based on the lessons learned. This will involve updating the BCs to represent the game's learning progression better. This will be done by dropping the current BCs and moving to a new set. This new set, described in the next Chapter, will be based on what we learned in Chapter 7 regarding what makes a level in *DungeonGrams* difficult to play. The unfortunate part of this change is that we will lose the ratings of difficulty and enjoyment of levels. However, we will use the linear regression models from the previous chapter to rate the new level segments in terms of difficulty and enjoyment.

2. The problem with random assignment is that it is, it turns out, random. You will likely get a reasonable, even distribution, but it is not guaranteed. Other approaches beyond simple randomization can get better results, like block randomization [51], stratified randomization [98], and covariate adaptive randomization [111]. Based on the work from [92], block randomization—see Section 9.1 for an introduction—would have been the best approach for

the study because the population size of the study is small to moderate, and we do not have to control for covariates. Therefore, the next chapter will utilize block randomization to ensure a better population distribution across the conditions.

3. The limited playtime will be addressed by adding a minimum playtime and increasing the pay. This, though, does come with a downside, which will be a reduced population size, further incentivizing block randomization for condition assignment.

Despite the problems, we were able to find significant statistical differences between the conditions, and were able to show that the MDP-approach was stronger than random. This enables us to answer **RQ2.2**: Is it better to optimize for difficulty, enjoyment, or both, in terms of player experience and dynamic difficulty adjustment, when assembling levels? The effect of optimizing for *difficulty* in this study was a higher likelihood of the player dying more often and slightly disagreeing with the statement that the game was too easy. The increased death count resulted in players seeing fewer levels. In contrast, optimizing for *enjoyment* resulted in fewer deaths and more levels explored. However, *enjoyment* was not significantly different statistically when it came to the game being easier than *random*, which is a bad result for a DDA system. When the rewards were combined, we tested two approaches: (1) switching between the rewards and (2) calculating the mean. We found that switching between rewards was very similar to *random* in terms of performance, but it received the fewest number of participants at only 36. Conversely, *mean* was rated to be less easy to play than *random*, had many levels explored, and players also disagreed with the statement that the game was too easy.

Chapter 9

Automatic vs. Handcrafted Markov Decision Processes

> Still looking him in the eye, I stepped off the edge of the roof.
>
> Patrick Rothfuss, *The Name of the Wind*

This Chapter is based on the paper, "Evaluating the impact of MDP-based level assembly on player experience" [15]. The authors were me and Seth Cooper.

The goal of this chapter is to answer **RQ2**: Does an MDP built automatically, a handcrafted MDP with designer-set rewards, or a static progression result in a better player experience? This chapter is the culmination of all the work done, because it essentially answers the simpler question: Does level assembly via a MDP for DDA work? Two games were used to answer **RQ2**, a roguelike and a platformer. The first was *DungeonGrams*, which has been discussed ad nauseam at this point in the dissertation. The second game used was *Recformer*. *Recformer* is a platformer where the player plays a rectangle—hence the portmanteau name for the game—and they have to collect every coin in the level to beat the level. (More details are available in Section 9.2.)

To answer **RQ2**, a player study was run for each game with four conditions: *r-mean*, *r-depth*, *static*, and *hand*. The first three were built with *Ponos*. *r-mean* is the default output of *Ponos*, and, based on the results from Chapter 8, the reward was set to be the mean of the expected difficulty and enjoyment of each level segment. *r-depth* replaced the reward with the depth of the level segment to encourage the MDP to get to the last level segment as fast as possible. *static* was a SLP, where a breadth-first search was run on the MDP to find the shortest path from the *start* level

segment to the *end* level segment. Finally, *hand* was a MDP built by hand rather than an automatic progression; additional software was required to make it feasible to make an MDP by hand, and the tool built is introduced in Section 9.4.

The two player studies run were very similar to the study run in Chapter 8. However, several lessons were learned from running that study, and changes were made accordingly. For example, the previous study had a bad distribution of condition assignments, and this was fixed by assigning conditions with block randomization; see Section 9.1 for more on block randomization. Section 9.3 has more details on the player study and other changes made.

We found that the SLP for *DungeonGrams* was too easy to beat and the SLP for *Recformer* 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 mPXI [74]. Further analysis was conducted based on the gameplay data, which yielded more statistically significant results. Overall, the handcrafted condition performed the best.

9.1 Background: Block Randomization

Block randomization [51] is a method for condition assignment [92]. The name is misleading because it sounds somewhat complicated when it is, in fact, very simple; the only method of condition assignment that is simpler is random assignment.

As a working example, let's say that we want to run a study with two conditions: condition A and condition B. Block randomization works by building a "block" of those conditions. A block, in terms of a program, is an array. So, we would have an array: `block = ['A', 'B']`. We could add an index (`block_index=0`). Then, when a condition was requested, we could grab the condition at the index, increment the index (`block_index = (block_index + 1) % block_array.length`), and wait for the next condition request.

There is a problem with the algorithm, though: it isn't random. The first, third, fifth, etc. participant would always be assigned condition *A* and every other participant would be assigned condition B. To resolve this, the array of conditions should be shuffled. Not just once, though. The block should be shuffled before any condition is assigned and after every time the index would overflow (i.e., remove the modulus operator from incrementing and then test: `block_index >= block.length`). This adds randomness to the condition assignment.

One last extension that can be added is k blocks. Rather than one condition per block, you can have k conditions per block. This adds additional randomness to the condition assignment.

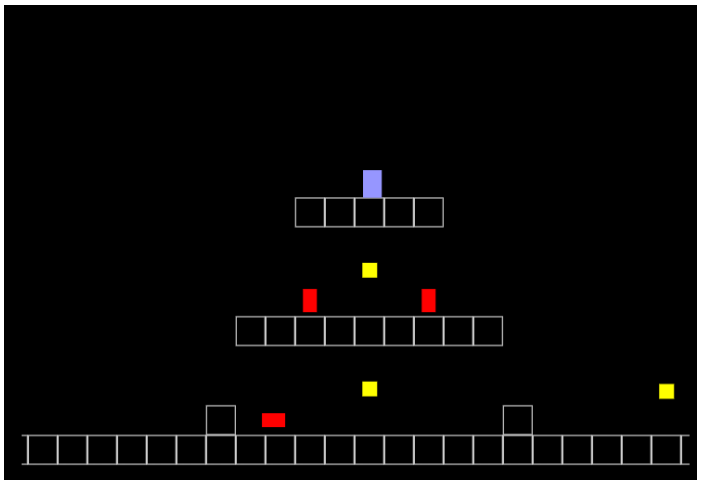


Figure 9.1: Example screenshot from *Recformer*. The player is the purple rectangle towards the top. Red rectangles are enemies that the player wants to avoid. Yellow squares are coins that must be collected for the player to complete the level.

In our working example, we know that conditions A and B will be assigned every two conditions requested. If we set k to three, then the block would triple in size, and we would no longer be guaranteed that the first two conditions would always be A and B. Instead, the guarantee becomes that the condition A will be assigned three times and the condition B will be assigned three times every six conditions requested. The total size of the block—`k*block.length`—should be divisible by the number of participants, else there is a risk of an unequal distribution.

The downside of using a method like block randomization, instead of random assignment, is that a server is required to maintain the state of the block array and index, and assign conditions to participants. For the two-player studies run in this Chapter, we used a server to assign conditions and store logs.[1]

9.2 *Recformer* the Game

Recformer, see Figure 9.2, is a platformer. The player can move to the left and the right and can jump. There are four important differences between *Recformer* and the standard platformer:

1. Unlike some platformers (e.g., *Mario* [131]), there is no way for players to jump on top of enemies to damage them. Instead, any contact between the player and an enemy is fatal to the player. (The other way for the player to lose is to fall through a gap in the level.)

[1] Code available on GitHub: https://github.com/bi3mer/go-log-study-server

2. *Recformer*'s jump is implemented differently from standard platformers. In most platfromers, the player's avatar must be connected to a ground tile to jump; essentially, something to jump off of. In *Recformer*, though, the player can jump while in the air. Therefore, a valid platforming puzzle in *Recformer* is to have the player fall until they go underneath an obstacle and then press the jump button to get to a higher platform on the other side of the obstacle.[2]

3. There is no sprint functionality like in other platformers, such as *Super Meat Boy* [182]. The goal is for the game to be less twitch-based than most platformers, allowing for more methodical gameplay.

4. The goal is not for the player to reach the end of the level to win; a level is beaten when the player has collected every coin in the level. Therefore, a level with no coins is automatically beaten. This could be addressed in the PLG portion of *Ponos* to only generate levels with at least one coin, but instead, padding was added to both sides of the level, so that there was a safe place for the player to start and so that there was always a coin at the end of the level.

To make the game more interesting to play, additional entities were added to challenge the player:

- Horizontal Enemy - This enemy moves horizontally and changes direction when it reaches the end of the level or a solid block.

- Vertical Enemy - This enemy moves vertically and changes direction when it reaches the top or bottom of the level or a solid block.

- Circle Enemy - This enemy moves in an oval shape and ignores obstacles like solid blocks.[3]

- Turret - The turret activates when the player is within ≈ 12 blocks of it, in terms of Euclidean distance. It then charges for 2.5 seconds and fires a bullet. The bullet is fired in the player's direction and continues in that direction until it comes into contact with any entity in the game or times out. Unlike every other entity in the game, bullets can destroy horizontal, vertical, and circle enemies.

[2] Originally, this behavior was the result of lazy programming on my part, but I found that I liked the behavior while testing and kept it in the game.

[3] For some reason, it feels like circle enemies are chasing you when you get close, but they aren't. Their movement is deterministic and not affected by the player.

- Laser - The laser has a similar behavior to the turret, except that instead of a bullet, it fires a laser straight up. The laser fills the space from the laser block to the top of the level or the first solid block. The laser does not harm enemies, but it will destroy the bullet on contact.
- Blue Block - On contact with the player, the blue block slows down the player's acceleration, and allows the player to jump again if they haven't already.

Recformer is implemented in TypeScript[4] and playable on the web.[5] The implementation includes an agent that can play the game and has code that shows how it interacts *Ponos*.

Because beating a level *Recformer* has nothing to do with how far to the right the player made it through the level, the completability agent does not use how far the player made it through the level as part of the percent completable return. Instead, it returns the number of coins that it was able to find in the level divided by the total number of coins.

9.2.1 A Note on Ponos for *Recformer*

One situation that can occur in *Recformer* is when a player reaches the end of the level, collects the coin that was added as padding, and the level doesn't end. This means that they missed at least one coin. As a result, the player has to backtrack through the level to find and collect the missing coin(s). A level where the player cannot backtrack may start as completable but become uncompletable based on the player's actions. This can result in a frustrating experience for the player. Using *Ponos* may exaggerate this because the MDP may serve the player an easier level, and this may feel to the player like they are being penalized.

Section 8.4.4 discussed how losing a level while API was the director could be frustrating. Specifically, the initial behavior was to remove an edge from the *start* node after losing; players incurred an additional penalty when losing. Given the inability to backtrack, the frustration could theoretically worsen. Therefore, the API director for *Recformer* does not remove edges until the player has failed at least two times in a row. This, though, can be configured. The number of player failures before edge removal is a way for designers to influence the player's experience, and the number should change based on the game and the designer's goals.

That being said, there was a way to fix the potential problem of it being impossible to backtrack to find a coin, which was to make the completability agent test that every coin was reachable from every other coin, with at least one coin being reachable from the player's start position. I

[4]Code available on GitHub: https://github.com/bi3mer/recformer

[5]Link to play *Recformer*: https://bi3mer.github.io/recformer/. The default version is the handcrafted condition.

did not opt for this approach, but it would be the correct thing to do if *Recformer* was to be released commercially (which it will not be, and it is open source for any to use in their own work).

9.3 Player Study Details

Two studies were run using methods approved by Northeastern's Institutional Review Board. One for *DungeonGrams* and one for *Recformer*. Besides the game the participant played, each study was exactly the same. The format for each study followed what was described in Section 8.4. Players were recruited via Prolific, they played the game, they filled out the survey (demographics, mPXI, and then three custom questions), and then they were shown their code to get their payment. There were, though, several differences:

- There were four conditions: *r-mean*, *r-depth*, *static*, and *hand*. (Each condition is described in detail in the sections that follow this one.)
- The study in Chapter 8 recruited a total of 250 participants, but had a low play time. One part of the fix was to reduce the number of participants to 40 per condition (160 total), and the pay was increased to three dollars per participant. The estimated time was increased to fifteen minutes for a pay of twelve dollars an hour.
- Rather than have a button that the participant could press at any time to exit the game, a timer was added at the top of the webpage for the game. The timer counted down from ten minutes, and when the timer hit 0, the participant could no longer play. This, though, made it so they could not exit the game early.
- The version of *DungeonGrams* used in Chapter 8 included a survey built into the webpage. To better match previous work with the mPXI [74], a change was made to link to a Qualtrics survey instead. Once the timer hit 0, the link to the survey was presented with instructions. After participants completed the survey, they were shown the code to complete their Prolific study.

For *DungeonGrams*, levels were built with 2 level segments. For *Recformer*, levels were assembled with 3 segments. The decision to use 2 level segments for *DungeonGrams* follows the same logic as what was presented in the previous chapter. 3 level segments were used for *Recformer* based on testing where 2 level segments felt too short to play and 4 too long. 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 has more influence over the levels assembled.

9.4 Handcrafted Condition

The *hand* condition was built by building an MDP by hand for both *DungeonGrams* and *Recformer*. This section discusses the motivation, goes over the level segments that can be built, describes how the level segments are used to build a digraph by hand with a tool, and then shows the MDPs built for both games along with some basic stats.

9.4.1 Motivation

A critical part of any procedural content generation system is *controllability*. Using *Ponos* means the entire MDP is built automatically, giving the designer less control over the player experience than with an SLP. The designer may not like this, but they may still want DDA via levels in their game. The alternative to using *Ponos* to create an MDP is to make an MDP by hand.

Levels do not have to be generated with Gram-Elites. Linking does not require level segments built by Gram-Elites. In fact, it is not strictly necessary that levels be linked at all. Designers can build level segments and decide which should connect with which. Rewards for level segments do need to be set via a player study. The alternative of using heuristics instead is not the only alternative. Designers can set the rewards themselves.

Building level segments, connecting them manually, and setting rewards by hand, though, is a lot of work, but making games is a lot of work. The best way to make this easier is by building and using tools.

9.4.2 Building Levels

Video game levels are typically built with a level editor [40, 71]. A level editor is a program for users to build levels for a game. Some level editors are simple, and some are complex. For levels in this dissertation, we can expect a grid-based layout in two dimensions. This means a simple text file where you have rows of characters, where each character represents a different entity in the game, with one character meaning a blank spot (nothing should go in row A, column B), similar to what is seen in the Video Game Level Corpus [176], is all that is required to represent

a level. Therefore, a simple text editor is more than enough for building levels in this work; it could be vi, vim, or nvim.

It is not, however, required to edit the text files directly. Tools like Tiled[6] or LDtk[7] are valid alternatives. You could also implement an editor of your own. But, regardless of how the levels are built, once they are made, they need to be used to build a digraph.

9.4.3 Making a Digraph

As a reminder, Chapter 5 showed how a digraph can be turned into an MDP. Therefore, the task here is to make a digraph with level segments that the designer built.

A digraph can be defined with a format like JSON, where each entry in a dictionary is a node. A node has a reward, a level segment (represented by a file name or path), and a list of neighbors. Editing the graph JSON file with only a few level segments is easy enough. However, it becomes unmanageable as the graph gets bigger. It can require debugging when there is a typo or a missing link. There can be leaves in the graph that break level generation since the graph is not fully connected. There can also be problems with rewards that are too big or too small that go unnoticed in a text file that is thousands of lines long. For all these reasons, a text editor is insufficient. The solution is a graph editor, see Figures 9.2 and 9.3 for examples of the graph editor built for this work.

The graph editor built for this work is called GDM-Editor.[8] GDM stands for graph-based decision making, and is a tool I built for making graph-based MDPs.[9] There is also a TypeScript version of GDM.[10]

GDM-Editor receives as a command line argument a path to a directory that contains a `graph.json` file and a directory called `segments`, which has a set of level segments built by the designer. Each file in the `segments` directory is a node in the `graph.json` file. The format of the text file for level segments does not have to be the already described text format, but there is a preview function that is only useful if the file contains something that is human-readable. When the designer adds a new level segment to the `segments` directory, a node is added to the graph and is visible in the editor. The designer can move nodes and connect them to other nodes. Each node also has a reward field where the designer can enter the reward.

[6] https://github.com/mapeditor/tiled
[7] https://github.com/deepnight/ldtk
[8] Code available on GitHub: https://github.com/bi3mer/GDM-Editor
[9] Code available on GitHub: https://github.com/bi3mer/GDM
[10] Code available on GitHub: https://github.com/bi3mer/GDM-TS

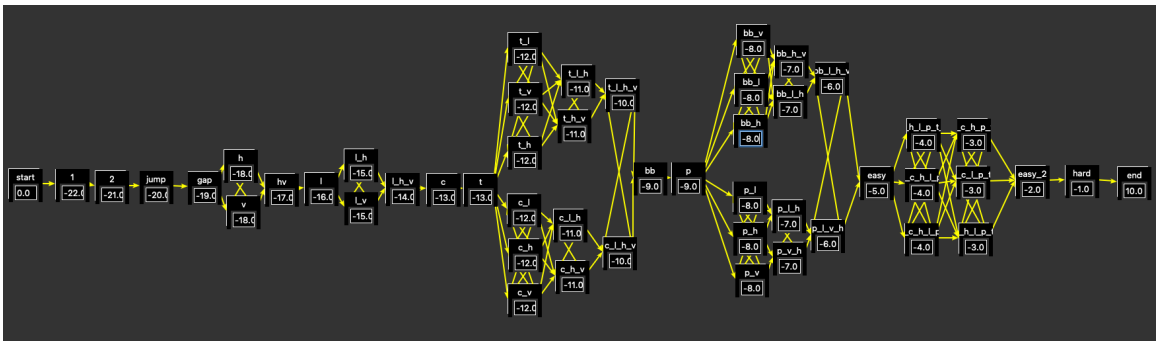


Figure 9.2: The handcrafted Markov Decision Process built for *Recformer* displayed and built in GDM-Editor, and it is used by the *hand* condition.

These functions allow for an iterative design process. The designer can build a few levels, connect them, test the progression, and refine them before moving on and repeating the process with new levels.

One problem found while using the editor was that it quickly 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 is 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 editor does not include any functionality for testing the edges. That is left to the developer. That, though, would become extremely tedious given any progression composed of more than a couple of level segments. To handle this, I wrote a script that used the agent built for *Ponos* to test every edge for both games. Both scripts are part of each game's repository, but are not directly part of the editor itself.

9.4.4 Graphs Built for *DungeonGrams* and *Recformer*

Figures 9.2 and 9.3 show the fully built progressions for *DungeonGrams* and *Recformer*. The MDP built for *Recformer* is composed of 49 nodes, with 138 handmade level segments. When the nodes are decomposed, there are 138 nodes and there are a total of 1161 edges. The path from the *start* node to the *end* node is 26 level segments long. The MDP built for *DungeonGrams* is

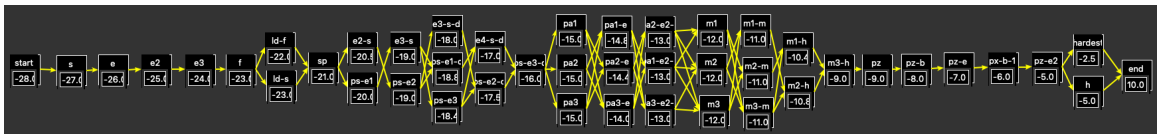


Figure 9.3: The handcrafted Markov Decision Process built for *DungeonGrams* displayed and built in GDM-Editor, and it is used by the *hand* condition.

composed of 44 nodes, with 102 handmade level segments. When the nodes are decomposed, there are 104 nodes and there are a total of 342 edges. The path from the *start* node to the *end* node is 27 level segments long.

The process of making both level progressions was iterative, and the primary goal was to introduce a concept and then ramp up the challenge. Then, I repeated this until every game element was in the final level. For *Recformer*, this could have resulted in a highly linear progression by showing the horizontal enemy, then the vertical enemy, then the circle, and so on. Instead, Figure 9.2 shows the alternative direction where the beginning is a very controlled experience where the player is being taught the very basics, like moving, jumping, and that they have to collect the coin. Then they are either introduced to a horizontal enemy or a vertical enemy. The theory here is that if the player understands one, the other is not that difficult to understand. Whichever they play against, the next node has levels that contain both horizontal and vertical enemies and are much more challenging. The next portion of the progression forces the introduction of circle enemies and turrets, and then there are two paths. Both paths heavily test either concept, but if the player struggles on one, the MDP will push the user to the other path. This theory is again repeated with blue blocks and platforming puzzle levels. The final section was made to be much more difficult and has every in-game element, but there are "easy" sections in between to give the player a break. This break also serves as a jumping-off point for the player that they are likely to always beat, and they will therefore not have to suffer too heavy a penalty when losing multiple times in a row.

The *DungeonGrams* progression was more difficult to make. Unlike in *Recformer*, there are not nearly as many mechanics and concepts to introduce to the player. The challenge in the game comes from different kinds of movement puzzles based on stamina and avoiding enemies. The challenge is not necessarily from the number of enemies, but from how the player moves around the enemies. If, for example, there is an enemy in a corridor, but the player can move forward without having to adjust, then that level is not as challenging as one where the enemy is placed such that the player has to change their path. The levels designed show different kinds of movement challenges. In *Recformer*, levels of the same type were always included in the same node. For *DungeonGrams*,

though, some versions of each level were more challenging than others. To take advantage of this, multiple nodes were added with names like `XX-1` and `XX-2`, and the rewards were updated to show this. For example, the last levels `hardest` and `h` have different rewards, because the `hardest` level is more difficult than `h`, but I still want the player to be able to finish the game if `hardest` is too difficult to figure out.

9.5 The r-mean Condition

This section describes how the *r-mean* condition was made and the parameters used for both *DungeonGrams* and *Recformer*. Further, some stats are given about the MDPs that were produced by *Ponos*. However, unlike in the previous chapters, there are no visualizations because more than two BCs were used for both games.

9.5.1 Gram-Elites

The configuration details for Gram-Elites below are very similar to what was given in Chapter 3. One difference between that work and this is that the resolution of the BCs was not configurable per BC in the previous work, but that has been changed in this work so that every BC defines its own resolution. For example, say a developer has a game where density is highly impactful on gameplay but not so much leniency; the developer of such a game may want a finer-grained resolution for density and a coarser resolution for leniency. Another change is to allow the designer not to have to specify a resolution. This allows for binary BCs like, "Does the level require mechanic X?" [100]. It also lets the designer return counts, like, "How many enemies of type X are in the level?"

9.5.1.1 *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. The right side padding added a portal to the bottom right corner of the level.

- *Behavioral Characteristics*:

  - Density: returns $min(1, solid\_block\_count/(0.75 * area)$. In words, this density is as described in the previous chapters, but the denominator is reduced by a quarter. This was done because there is little difference between a level that is eighty percent dense and ninety percent, which allows more levels to be found in the bottom three-quarters of density.

    * Resolution: 20.

  - Enemies: returns $enemy\_count * 8$.[11]

  - Switches: returns $switch\_count * 4$.

9.5.1.2 *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. Since there were no previous works, there were no additional levels as input. As a reminder, this was 138 levels as input.

- *Generated segment size*: 15 columns.

- *N-Gram Generation Iterations*: $1,000$.

- *MAP-Elite Iterations*: $10,000$.[12]

- *Padding*: A padding of two columns was added to the left and right sides of the level segment. The columns were empty, besides a solid block at the bottom. The farthest right column also had one coin.

- *Behavioral Characteristics*:

[11] The reason for the multiplication is discussed in Chapter 10.
[12] The reason for fewer iterations when compared to *DungeonGrams* was due to the speed of the completability agent.

- Inverse Density: returns $1 - density(level)$. Levels in *DungeonGrams* are more challenging when the level becomes dense and the player has to interact with enemies, but the same is not true for *Recformer*. For *Recformer*, the inverse of density was better suited for it because, as a platformer, the less dense levels resulted in more opportunities for the player to fall and lose.
  - 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 $blue\_block\_count$.

9.5.2 Linking

For *DungeonGrams* and *Recformer*, empty links were allowed; an empty link is a link with no level slices. The maximum link length for *DungeonGrams* was 2 and *Recformer* was 1. The small link length would have been a problem for *DungeonGrams* if structures were kept, but they were removed for convenience. *Recformer* had no in-game structure, and longer link lengths didn't appear to make it more likely that an unsolvable level would become solvable.

The linking step is the step that takes the longest in *Ponos*. There are two reasons. First, the size of the level being tested by the completability agent is double the size of the generated level segments, which increases the solve time per level. Second, the number of possible edges to test can be at most 2 if there is one BC, 4 if there are two, 8 if there are three, and so on. This leads to a large number of levels that have to be tested when you take into account the number of elites per cell. Further, there tend to be levels that cannot be completed. Levels that cannot be completed are levels where A* will perform slowly, and this triggers the linking algorithm to try adding linking slices. This can lead to many failed solve attempts by the completability agent, which results in a long run time for *Ponos*. This was why a low link length was used.

9.5.3 Reward

After every link was built and the graph was pruned such that every node was reachable from the *start* level segment and could reach the *end* level segment, the reward was assigned.

For *DungeonGrams*, the reward was assigned by using the linear regression models built for predicting difficulty and enjoyment of a level segment and then taking the mean of the two. The model for enjoyment used *proximity-to-enemies* as input. The best performing model from Chapter 7 also used the computational metric *path-nothing*. This model was not used to better match the reward used by *Recformer* (see the paragraph below) so that the studies would be more similar.[13] The model for difficulty used *jaccard-nothing*, *proximity-to-enemies*, *stamina-percent-enemies*, and *density* as input.

The models were not used for *Recformer*. The main reason was that *Recformer* was a platformer, and the models were built for roguelike. There were, though, two technical problems as well. The first problem was that *Recformer* had no stamina mechanic, and the BC *stamina-percent-enemies* could not be replicated. The second problem was that *Recformer* was a non-grid-based movement game, and it was unclear what the best way was to use Jaccard similarity [86]. The player's position could have been rounded to the nearest integer, but then we would have had to figure out the correct time step or frame for when the player's position should be captured. So, Jaccard similarity was dropped.[14] To mimic the *mean* reward, the remaining BCs were used.

Enjoyment for *Recformer* was the output score of *proximity to enemy*, using every frame in the simulation. *path-nothing* was not used for *Recformer* because (1) removing coins would make the level instantly solved, and (2) the alternative of *path-no-enemies* was not used once to predict enjoyment. 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. This is because a dense level affords the player less area to maneuver. This u-curve density function was calculated with: $u\_density(lvl) = (solid\_blocks(lvl) - 0.5 * area(level))^2/(0.5 * area(level))^2$.[15] The mean of the two scores

[13] The coefficient of *path-nothing* was -0.017746. The product of the coefficient and the result of *path-nothing* was always close to 0, and so *path-nothing* was relatively unimportant to the calculation when compared to *proximity to enemies*, which had a coefficient of -4.0725 for the linear regression model that took both as inputs.

[14] The coefficient of *jaccard-nothing* to predict difficulty for *DungeonGrams* was -0.9590, which was large enough that it was not removed from the model used by *DungeonGrams*. The same was true for *stamina-percent-enemies*, which had a coefficient of 0.6253. As a result, some dissimilarity in how the reward was calculated for the two studies had to be accepted.

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

was then calculated, and that was the reward for every node in *Recformer*.

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 primary motivation was to set all the rewards to be negative, while maintaining order. Positive rewards with an infinite time horizon MDP result in the levels generated never approaching the *end* state, which is inappropriate for this context. There are alternative approaches to the function used. For example, the max reward could have been subtracted from every designer reward. However, the size of the negative rewards results in different kinds of behaviors from the MDP. If the rewards are too negative, then the MDP will essentially behave like a breadth-first search, and find the shortest path to the *end* node.

9.5.4 Final Markov Decision Process

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.

9.6 The r-depth Condition

This condition used the MDP built by *Ponos* for the *r-mean* condition. The designer reward, though, was changed from what was described in Section 9.5.3 to be based on the node's depth, or distance from the *start* node. The update made use of 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 is static, so when edges were added or removed, R_D was not updated.

One goal for this condition was to test whether it was necessary to craft a complex reward function when something simple like depth was all that was necessary, while still pushing the player towards the last level.

9.7 The Static Condition

The final condition not yet described is the *static* condition. This condition mimics an SLP, which, as a reminder, is an unchanging level progression. The player plays a level until they win, and then they move on to the next level. There were three options for how this condition 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.
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 me to make it, but that again opens up the results to the criticism that if the SLP condition performed poorly, it was because the designer (me in this case) did a poor job building the SLP. 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.

The SLP 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 nodes. Note, this ran on the deconstructed graph, so we aren't randomly selecting from a set of elites generated by Gram-Elites; the player only saw one level segment per index in the array of nodes. 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. In the case of *DungeonGrams*, this was two. In the case of *Recformer*, this was three. If the player lost, then the index would not be incremented, and the player would play the same exact level again.

The path length for *Recformer* was made up of 23 level segments. Meaning, the player had to beat 8 levels in a row without failing to win. For *DungeonGrams*, the path length was made up of 26 level segments, which resulted in the player having to beat 13 levels in a row without failing to win.

9.8 Analysis Method

The analysis conducted was similar to that in the previous chapter. Analysis of all Likert data was performed with the Kruskal-Wallis test [119]. This was done because ANOVA [169] assumes continuous 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 [48] with a correction via the Holm method [80] to find any statistically significant differences between groups.

For variables that were numerical, they were first analyzed to see if ANOVA [169] could be run. This was tested by running Levene's test [57] to see if the distribution was homoscedastic

and the Shapiro-Wilk test [155] to see if the data was approximately normally distributed. If passing, then ANOVA was run. If the p-value was less than 0.05, then post-hoc analysis was done with Tukey's test [188] and a Bonferroni adjustment [8, 19]. For variables that failed Levene's test or the Shapiro-Wilk test, the Kruskal-Wallis test [119] was run. For variables that had a p-value < 0.05, a post-hoc analysis was run with Dunn's test [48] with a correction via the Holm method [80] to find any statistically significant differences between groups.

9.9 Results

9.9.1 *DungeonGrams*

The median time per participant was 13 minutes and 24 seconds, which includes 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—this means that 161 in total were recruited by prolific, but one dropped out of the study—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 9.1 shows the results from the mPXI [74] and the three custom survey items. Only two items had statistically significant results, and neither was on the mPXI; meaning that the player experiences were relatively similar 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 compact letter display showed no major difference between the conditions. The p-values between groups are shown in Figure 9.4.

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-mean*, 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.

	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 9.1: 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.

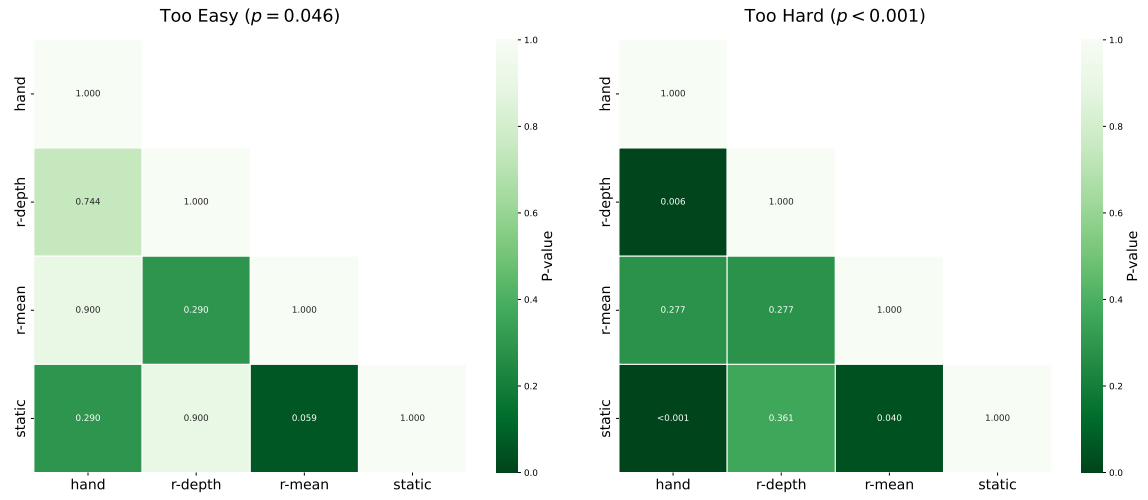


Figure 9.4: P-value matrix heatmap from Likert questions with a p-value < 0.05 for *DungeonGrams*.

	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 9.2: 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.

Table 9.2 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 "Time Played" 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 Played" shows how long the player played *DungeonGrams* with their assigned condition.

For "Time per Player," we can see that *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. Specifically, 70% of the *static* players beat the game—26 of the 37 players. As a result, these players did not end up having to play the full ten minutes. What makes this perhaps more interesting is that we would expect the scores for "Challenge" and "Too Easy" to reflect so many *static* players winning, but they don't. "Too Hard," though, does 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" had two groupings. The *static* condition was in its own group with the

least amount of levels played. This was due to players beating the game. The other group contained all the dynamic conditions, and there were no statistically significant differences between them.

"Levels Won" was interesting because there was a statistically significant difference, but the groups were all the same, with *static* having on average the most levels won and *r-depth* the least.

"Levels Lost" showed a significant statistical difference between the dynamic conditions and the static condition. The dynamic conditions had about ten more levels lost on average per player than the *static* condition. The difference is due to the players beating the game in the *static* condition. This, again, raises questions about what exactly players consider "challenge" to be. Based on these results, the expectation would be that players rated the dynamic condition as more challenging than the *static* condition. Perhaps with a direct comparison, they would have.[16]

The last two, "Lost by Enemy" and "Lost by Stamina," have two groupings. The separation is between the dynamic conditions and the *static* condition. The difference, for the most part, can be attributed to the number of levels the players played. There were no statistically significant differences between the dynamic conditions.

9.9.2 *Recformer*

The median time per participant was 14 minutes and 40 seconds, which includes time playing the game and filling out the survey. This yielded 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 9.3 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

[16] From playing all the conditions myself, I certainly would say that the static condition is much easier to play. The majority of levels are dense without enemies. Only at the end does the player have to deal with enemies, and the puzzles are not all that difficult, similar to the example given in Chapter 10.

	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 9.3: 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.

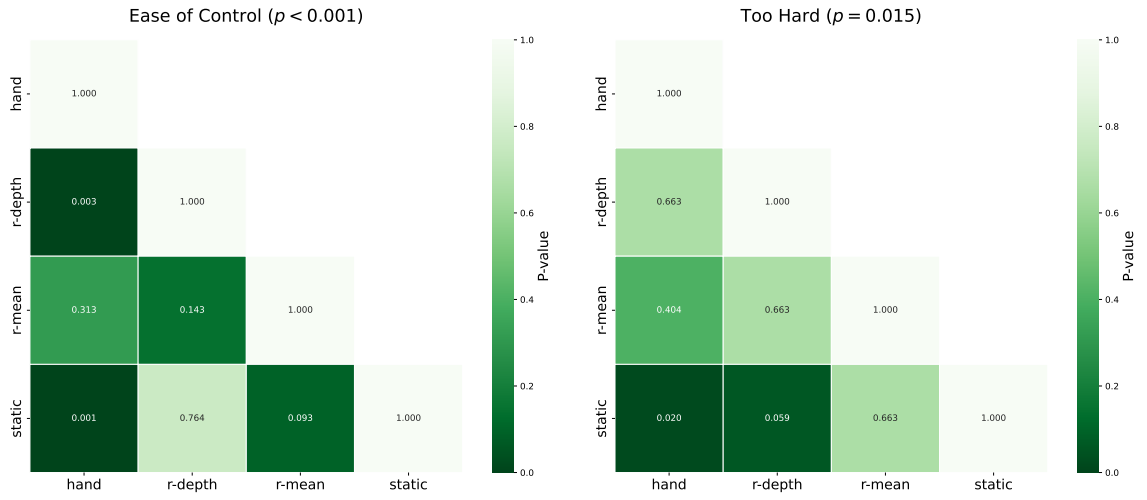


Figure 9.5: P-value matrix heatmap from Likert questions with a p-value < 0.05 for *Dungeon-Grams*.

	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 9.4: 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.

were less than 0.05. Figure 9.5 shows the p-value heatmaps for both.

"Ease of Control" 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." The p-value heatmap in Figure 9.5 helps show that there was a statistically significant difference between *static* and *hand* and *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 9.4 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 9.4 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.

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 3.737 levels beaten per participant. The majority of participants in the *static* condition got stuck and could not progress.[17] And recall here that "Challenge" from the mPXI showed no statistically significant differences amongst the conditions, and that participants in the *static* condition only slightly disagreed with the statement, "The game was too easy to play."

This takes us to "Levels Lost." There were three groups. The first group contained *r-depth* and *r-mean*. The second had *hand* and *r-mean*. The third and final group was made up of only 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. One thing to consider here is the number of levels played, where there were no statistically significant differences between the conditions. So, while some players played more and some players played less, there was not enough of a difference to have a major effect on the number of wins and losses. The *static* participants simply played more levels and lost more. This, though, did not appear to have a negative effect on the player's experience, as discussed above.

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.

9.10 Answering RQ2

The goal of this section is to answer **RQ2**: Does an MDP built automatically, a hand-crafted MDP with designer-set rewards, or a static progression result in a better player experience?

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

[17] 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.

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, with both groups found to be near the "Agree" response. Given the context of **RQ2**, which is about trying to identify the best player experience, these results are unfortunate. The ideal result would have been "condition X resulted in better scores than every other condition." However, there is more to a player's experience than what they report.

One such factor is whether the player beat the game. For *DungeonGrams*, 26 players in the *static* condition beat the game, but no players in the other conditions beat the game. Based on this, the SLP for *DungeonGrams* was too easy, which is one of the two distinct problems with SLPs. The other problem is when an SLP is too hard. 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. Level assembly via an MDP, though, clearly solved this problem by serving harder levels for *DungeonGrams* and easier levels for *Recformer*. The player experience from the lens of wins and losses was clearly better when players were not playing an SLP.

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. Again, there is nothing really to go by when examining the player experience based on the mPXI. 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. However, this is not enough to go by. It is worth noting that the automatically gener-

ated conditions have resulted in comparable player experiences to the handcrafted progressions for both games, and that there were no statistically significant differences between them and the *hand* condition for the gameplay data in *DungeonGrams*.

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 too low. Compare this to *DungeonGrams*, where the win rate for *hand* was 41.41%, for *r-depth* was 42.82%, and for *r-mean* was 41.49%. The structure of the digraph determines how successful the MDP will be, and the structure of the BCs for *Recformer* needed more work.

This takes us to the final answer to **RQ2**. From the win rates and the lack of differences in the player experience, it is clear that *hand* was the better condition for *Recformer*. Therefore, the answer to **RQ2** is that a hand-crafted MDP with designer-set rewards resulted in the best player experience.

9.11 Markov Decision Process Rewards

One of the questions relevant to all this work was: How important are the rewards in the MDP? In other words, is it necessary to spend a ton of time making a complex reward function to approximate something seemingly abstract like difficulty or enjoyment? While there were minor differences between *r-depth* and *r-mean*, 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. This is discussed more in the Section below as well as in Chapter 10.

9.12 Unexplored Areas of Future Work

- The ten-minute countdown helped improve the number of levels played, but didn't guarantee it; there were multiple players for both games who played under five levels. What wasn't considered, though, was that the analysis of time played became, for the most part, unimportant.

In the last study, it was used as a partial measure of engagement; more playtime equals better condition. An alternative approach that may have been better was to set a minimum play time and a maximum play time. Once the minimum play time was reached, the participant could quit, but they could also continue playing until the maximum play time was reached. In this study, the duration could have been a minimum of five minutes and a maximum of ten.

- The work so far has treated R_D as a static table, and this was true for the *r-depth* condition. An interesting test, though, would be to update R_D whenever an edge was removed or added to the MDP. This would (1) create a more accurate depth-based reward and (2) provide an interesting initial testbed for modifying the designer reward.

- The key factor that determined the success and failure of *r-depth* and *r-mean* was the BCs used by Gram-Elites to generate a structured population. A tool could analyze the BCs prior to running by analyzing the n-gram. If properly done, this tool could, for example, estimate the likelihood that a dense level with a bunch of enemies would be generated. Alternatively, Gram-Elites could be dropped and replaced with another tool like Sturgeon [34, 35] to generate levels that fit within a bin based on BCs. Sturgeon generates levels by solving a constraint satisfaction problem. Alternatively, a tree search through the n-gram [174] to find levels that match specific BCs could also be used.

- Given the success of the *hand* condition, more work should go into developing GDM-Editor and improving it for others to use. A study could be conducted to test the designer's experience with it, and the results could be used to further improve the tool. One immediate improvement would be to add functionality for editing levels directly within the editor.

- Using a breadth-first search to find the shortest path from the *start* node to the *end* node for the *static* condition was perhaps too simple an approach. An alternative that may have yielded a more balanced result for both user studies would have been to weight the path based on the rewards in R_D.

- It would have been interesting to include the *mean* MDP from the study in the previous Chapter as a condition in the *DungeonGrams* study. That would have allowed for a direct comparison to see what the changes in the MDP resulted in. As it is, any comparison between the two datasets would be flawed due to being run in different studies and there being differences in how each study was performed, specifically the ten-minute time requirement.

- The custom questions for "The game was too easy to play" and "The game was too hard to play" were, in hindsight, not the best questions to ask, when what we were really interested in was whether the *levels* were too easy or too hard. Work could be done to develop a series of statements that players could respond to on a 7-Likert scale to better understand how the levels themselves affect the player.

9.13 Conclusion

The player experience, based on the mPXI, 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 Challenge score worse than conditions with a lower win rate—the other three had win rates around 40%—because the levels would be too easy since players kept winning, 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 while playing.

Despite this, we were able to answer **RQ2**: Does an MDP built automatically, a hand-crafted MDP with designer-set rewards, or a static progression result in a better player experience? The *static* condition was the worst performing, with players beating most levels and almost never losing in *DungeonGrams*, but making almost no progress while playing *Recformer*. With static removed, that left the three MDP-based conditions. Finding the answer relied on the fact that since the player experience was so similar, there was no choice but to analyze based on the experience of losing and winning. For *DungeonGrams*, there was little to no difference, but *r-depth* and *r-mean* for *Recformer* both had much lower win rates—28.66% and 33.60% respectively—than *hand*, which had a win rate of 49.65. Based on these results, and a subjectively desired win rate for the games, the only answer was that *hand* was the best overall condition. The answer to **RQ2**, therefore, was that a hand-crafted MDP with designer-set rewards resulted in the best player experience.

Now that **RQ1** and **RQ2** have been answered, that leaves one last part: the conclusion.

Part IV

Conclusion

Chapter 10

Working with *Ponos*

> He'd tired of marvels: of their cost in
> human warmth. His wish was for **oblivion**.
>
> Alan Moore, *Miracleman*

This Chapter is similar to Chapter 6, except instead of giving a technical guide on how to run *Ponos*, the goal of this Chapter is to help you decide whether *Ponos* is right for your game by giving practical advice on using the system as well as some of the problems you may encounter. The first section covers a topic which is essential to using Gram-Elites, which is the selection of the right n for the n-gram. There is a whole paper [43] on finding the right n for n-gram level generation that is also relevant. The second section is on the structure of the digraph that is built from Gram-Elites. This section includes an example that is worked through, and makes an argument for why BCs can be a problem, which may influence your opinion on whether *Ponos* is right for your game. The final section before the conclusion discusses how you can use the reward of every node in the designer reward table to influence the progression that the player will experience.

10.1 The Right *N*

Finding the right size of n for the n-gram used by Gram-Elites has already been discussed in Chapters 2 and 3, but it bears more discussion for those who want to ask the question, "How do I find the right n if I want to use *Ponos*?" The simplest and best answer is to test multiple values of n by generating levels with an n-gram, and then pick the n with the results that you think are best. However, there are some things to keep in mind.

First, the smaller the n, the more diverse the output can be [43, 91]. This is because the larger n is, the more priors will be required to obtain an output. Put more concretely, if you have a bi-gram but for some reason wanted the behavior of a uni-gram for a game with 10 possible level slices, then you would need to provide 10^2 unique examples for all connections to be possible. If you had a tri-gram, this would go up to 10^3. The larger the n, the smaller the possible output set will be without increasing the size of the input dataset. This is a great feature because it allows the right size n to generate levels similar to those seen in the input dataset, while still being novel. When choosing n, developers should side towards the smallest n that maintains the look and feel of their game because this allows for the greatest diversity of output, which is essential for Gram-Elites to work well.

Second, the n-gram should at least be a bi-gram if your game has any in-game structures that are more than one level slice long. There are cases, though, where n may need to be larger. Say we have a game where a structure was four level slices long and each level slice was unique; a bi-gram would always generate that structure. However, if the structure was four level slices long and was made of duplicate level slices, a bi-gram would *not* always generate that structure. For example, imagine we represented the level slices with characters and the structure was `[a, a, a, a]` and there was an input level like `[z, a, a, a, a, b]`. A valid output for a bi-gram given the input level is `[z, a, b]`, an invalid structure. The way to fix this is that the n-gram should be at least a four-gram.

The downside of using an n-gram is that it discourages designers from building large structures with repeating characters due to needing to use a larger n. This is because the larger the n, the more diverse the input has to be if you want the n-gram to output level segments that are not exact repeats of what was in the input dataset.

10.2 The Digraph

Ponos outputs an MDP, and that MDP is structured as a digraph, and that digraph is ordered based on the BCs that were used by *Ponos*. BCs are one of the two ways that the designer can control the progression of their game if they use *Ponos*.[1] Rather than have an abstract discussion of BCs, let's use the concrete example of building the *auto-r* condition for *DungeonGrams* that was shown in Section 9.5.1.1. The BCs used were the count of enemies (multiplied by 8), the count of switches (multiplied by 4), and density (where area was multiplied by 0.75 and the max output

[1] The other way for designers to control the progression is by changing the designer reward table.

was 1). Recall from Chapter 3, the set of BCs for *DungeonGrams* was density and leniency with a resolution of 20 for both. A lot changed.

The reason why this change was necessary is already explained in the conclusion of Chapter 8. The initial progression built for *DungeonGrams* wasn't great. One of the first levels that a player could play was too difficult. There was no buildup to the puzzle to help the player learn the game. However, that does not explain how I ended up with the final list of BCs.

I tried many different sets of BCs before I found the final set that worked, and one of the problems I ran into was the use of density. The input levels made it very likely for Gram-Elites to be able to generate levels of all kinds of density. However, it was less likely for a level that was very dense to also have a bunch of enemies and switches. The result was that *Ponos* generated progressions that were mainly just increasing density, with very few instances of enemies and switches. It made for a very boring experience with no challenge.

One of the first solutions I tried was to reduce the resolution of density. However, this created its own problems. Most level segments could only realistically have 3-4 enemies at most and 1-2 switches. A resolution of 4 for density would result in a Gram-Elites grid with the dimensions $4 \times 4 \times 2$ with a path length from the start to end node of eight. Since two level segments were used per level, the minimum number of levels to play to beat the game would be four. That is not a long enough progression; both in terms of the time it requires to beat and, more importantly, in terms of the time it takes for a player to learn how to play *DungeonGrams*.

That is where the idea of multiplying different counts came from. The exact numbers of 8 for enemies and 4 for switches were found experimentally, but they also reflect the final resolution used for density, which was 20. 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. For example, if there were 4 enemies, then that level would be placed in a cell at 32 for the enemy count dimension, which will be later in the progression than the max density, which is cell 20 in its dimension. What this represents is an increased importance placed on enemies being towards the end of the progression. The same is true for switches but to a lesser degree, since their count is multiplied by 4.

There is one other alternative, which is instead of using a tightly constrained set of BCs, you could try a large set instead. This would resolve the problem already mentioned of a short progression given a small resolution for density. This, though, has its own set of problems. The most important of these is that Gram-Elites will not find a usable level for every possible bin. Consider, for example, using eight different BCs and let's say we have a resolution of 10 for each

BC. This means we are trying to find 10^8 (i.e., one hundred million) usable levels. The curse of dimensionality will become a real problem if you try to have too many BCs, and the result will be that many levels generated are not able to be linked to the start and/or end node. As a result, you are left with a bunch of wasted levels and a progression that probably does not build the more complicated levels that make your game interesting.

For example, I originally wanted to use a BC which was based on the amount of food in a level, such as `max(10-food_count, 0)`. A harder level is a level with less food in it, so the order is reversed so that later levels in the progression will have less food. However, a level where you have to pick up food is harder than a level without food because the player can be forced to deviate from their route. A more complex BC would use an agent to try and solve the level if the food was not in the level [36]. Then I could have counted the number of essential foods and multiplied that by a number. This, though, would be very slow BC to calculate for any level with essential foods because agents are at their slowest when levels are not solvable because they have to explore all the possibilities to declare that a level cannot be beaten, whereas to declare that a level can be beaten they only need to find one possible solution. All these considerations were unimportant, though, because of the curse of dimensionality. Gram-Elites did not find enough levels to make a large enough progression graph. I could have (1) tried changing the resolution for density and the constants for multiplying the counts for enemies and switches, or (2) increased the number of iterations that Gram-Elites runs for. However, I ultimately settled and dropped the food BC. Further, there is the other consideration of the amount of data that you are asking a user to download before playing. An alternative is to serve level segments via a server to reduce the download size.[2]

10.2.1 The Problem with Behavioral Characteristics

Let's stick with the *DungeonGrams* example where the goal was to create a progression where levels with a high density score were deemed more challenging than levels with a low density score. This assumption came from playing the game hundreds, if not thousands, of times, and it is mostly correct, but there are exceptions.

[2] *Recformer*, for example, has an initial stutter when the page loads because of the size of the MDP that was generated.

```
level 1          level 2          level 3
@----X-----      @---------       @XXXXXXXX
-----X-----      ---XXXXX--       -XXXXXXXX
-----X-----      ----#-*X--       -XXXXXXXX
-----X-----      -------X--       -XXXXXXXX
-----X-----      ----------       -----#---
-----#----O      ---------o       --------O
```

If you look at the levels above, which level do you think would be the hardest? If you haven't played a lot of *DungeonGrams* before, I'm betting that you would have guessed that level 3 would be the hardest to play if I hadn't led you to think otherwise with the above paragraph. Level 3 has an enemy (`#`) that is directly in the path of the player (`@`) to the portal (`O` for open portal and `o` for closed portal), and the player has no room to maneuver around the enemy without being in at least one block of the enemy. The same problem exists in level 1, but the player has much more room to maneuver the enemy away from its starting place. However, level 3 is the easiest of the bunch. Take a look at the solution path:

```
level 3 solution
@XXXXXXXX
.XXXXXXXX
.XXXXXXXX
.XXXXXXXX
.----#---
........O
```

This path (`.`) does not include any tricks like backtracking or pausing a turn. The player presses down until they reach the bottom of the level and then presses right until they reach the portal—no interactions with the enemy required. Levels 1 and 2, though, require backtracking and pausing for a turn for the player to win.

The goal of all this is not to make the argument that using density as a BC was bad. It is essential that levels feel distinct as the player progresses through a game, and density is a straightforward and visual way to achieve this—this claim is not verified, but it is something that I think makes sense in theory. However, the way a level looks does not directly translate to a progression in terms of challenge. We have, in effect, competing interests when using BCs to build a progression.

An alternative that was explored, but quickly thrown out, was to use the heuristic *proximity-to-enemies* as a BC. However, *proximity-to-enemies* was a poor fit for two reasons. First, as we have already seen with level 3, simply being near an enemy does not necessarily mean that the level is inherently difficult. Second, an agent will solve a level, but the solution it finds may not be the "easiest" solution. For example, imagine we remove the switch ($\star$) from level 2. The level could be solved by going past the enemy, but it could also be solved by going to the right and down, avoiding the enemy completely. The agent may find an easier solution, but it may also find a solution that moves directly next to the enemy, and the result will be a high *proximity-to-enemies* score. These two reasons are why *proximity-to-enemies* performed poorly as BC because a level that was supposed to be later in the progression, and therefore more challenging, could end up being very easy to beat.

As a result, using BCs to structure a progression is feasible, but it can be a challenging experience. In my experience making the progression for *DungeonGrams*, I must have made over thirty different versions. This is why a fast completability agent is essential. Otherwise, iteration would have been impossible, and without iteration, I would have never found a set of BCs that worked.

10.2.2 A Hybrid Approach

Ponos has been defined almost as an all-or-nothing model. It either does everything for you or you have to do everything yourself. This does not have to be the case. There is a middle ground. In fact, I'm sure that there are multiple middle grounds, but I'm going to discuss one specific middle ground.

The question/scenario I want to address is: "I want to use *Ponos*, but I want the digraph to be a specific structure based on what I have put in the game. How can I do that?" To answer this question, let's use *Recformer* as an example case. You may recall the *hand* condition, and it had a progression structure built into it, see Figure 9.2. It began with an in-game tutorial to teach the player the basics, then it introduced two kinds of enemies, and then it introduced more concepts in pairs. *Ponos*, as described, would have trouble building a progression like this without a significant amount of work to get the BCs just right. Further, the iteration time on such a progression would be slow because there would need to be many iterations for Gram-Elites to build all the levels.

The easier way a structure like the one built for *Recformer* could be built with *Ponos* is by running *Ponos* in parts. Let's work through the example just a bit so everything will be clearer. The

progression for *Recformer* begins by introducing basic mechanics to the player, such as jumping gaps and collecting coins. We could build two simple BCs, one for coins and one for jumping.[3] Once we had the BCs set up, we would want to use a small resolution for both, then *Ponos* could run and output a digraph. The next portion of the progression in *Recformer* introduced horizontal and vertical enemies. We can again build BCs, perhaps requiring more dense levels or fewer, depending on our preference, and then we could run *Ponos* again to output a new digraph. This process of creating small digraphs could continue until the whole progression was done. Then, the graphs could be combined into a single large digraph, and we would have a custom digraph built using a hybrid approach.

10.3 Reward

Chapter 9 showed that the designer reward table (R_D) did not have a huge impact on the player's experience. The structure of the digraph and the levels contained were far more important. However, the wrong reward could have a huge impact. For example, if all the rewards were positive and unchanging, then the player would never reach the end of the game because the policy built by policy iteration would not risk the player reaching a terminal state. This shows the other problem of all positive rewards besides the *death* state: the policy would never risk challenging the player. A reasonable conclusion from this is that the rewards must be negative, but this is not necessarily correct.

The work done in Part III used negative rewards for the above reason, but also because they are a good starting point. There are a lot of areas for experimentation when it comes to the designer reward table. For example, the negative reward strategy yields policies that favor reaching the terminal state,[4] but the designer may want the player to experience more of the game and what it has to offer. One way to address this is by changing how the reward of a node is updated. Initially, it could be a positive value, and only become negative once a player has played the given node's level segment or even beaten it. This would result in policies that push the player to explore all kinds of levels that they may not have played otherwise. One potential problem, though, is that it would not only force them to play some level segments, but also force them to essentially play all of them before they could reach the end node.

[3] The jumping BC should make use of the playthrough done by the completability agent to test whether a jump was required, and, if so, how many.

[4] If the rewards are negative enough (relative to the positive reward of the terminal node), then the policies built will throw caution to the wind and try to get the player to the end of the game as fast as possible.

There are some strategies to consider that would fix this problem. Random nodes could be selected to be positive. Better yet, the designer could use GDM-Editor, see Section 9.4.3, to choose which levels they wanted to start with positive rewards. There are other options as well, but the point I would like to make is that few things are set in stone with how a reward should be updated. The strategy I used is just one strategy.[5] In some cases, my strategy may even be a bad one, such as a node with a level segment that introduces the player to a new mechanic. It may be better for the update strategy for that particular node to be static, so that the penalty of going back to show the player a mechanic is not too heavy.

Different nodes can update in different ways if that is what the designer wants. This could be a lot of tedious work, but it could also be automated. There are other reward update schemes than multiplying by the visit count or making the reward unchanging. For example, one approach could be to base the reward on the node's win rate; the higher the win rate, the lower the reward. There are many options, and anyone who is considering *Ponos* can and should experiment before coming to a decision on what is right for their game.

10.4 Conclusion

If there is any conclusion that you should take from everything you just read, it is that nothing about using *Ponos* or the MDP is set in stone. Whether you want to figure out new BCs, reward schemes, node types, or more, everything is valid if it works for your game. You can make a totally custom structure for the progression of your game by using *Ponos* to generate parts of the graph and put them together. You can make an entire progression and have *Ponos* fill in parts that you don't want to bother making yourself. You can change the level segments after Gram-Elites made them. You can add nodes in between nodes to add a breather or make a difficult transition. There is no right way to use *Ponos* and its output, but there are plenty of wrong ways, and, hopefully, the discussions in this Chapter will help you avoid the wrong ways if you decide to try and use *Ponos* for your game.

[5] As a reminder, the strategy for node rewards updates was to multiply the designer reward by the number of times it was visited. This forces the reward to become increasingly more negative.

Chapter 11

Conclusion

> Beyond that I cannot guess what became of him. Even little Niggle in his old home could not glimpse the Mountains far away, and they got into the borders of his picture; but what they are really like, and what lies behind them, only those can say who have climbed them.
>
> J.R.R. Tolkien, *Leaf by Niggle*

Two research questions were answered in this dissertation, and each research question had two sub-research questions:

- **RQ1**: How can usable levels be assembled from level segments for dynamic difficulty adjustment?
  - **RQ1.1**: How can usable level segments be generated within an organized structure applicable to dynamic difficulty adjustment?
  - **RQ1.2**: How can usable level segments be linked together to form levels?
- **RQ2**: Does an MDP built automatically, a handcrafted MDP with designer-set rewards, or a static progression result in a better player experience?
  - **RQ2.1**: How well do computational metrics approximate difficulty and enjoyment?
  - **RQ2.2**: Is it better to optimize for difficulty, enjoyment, or both, in terms of player experience and dynamic difficulty adjustment, when assembling levels?

Each sub-research question and major research question was answered within a chapter:

- Chapter 3 answered **RQ1.1** by building a PLG method called Gram-Elites, which upgraded MAP-Elites with n-gram-based genetic operators. A study was run with three different games, and Gram-Elites was compared to base MAP-Elites and an n-gram level generation approach. Gram-Elites outperformed both for all three games.

- Chapter 4 answered **RQ1.2** by showing how level segments can be linked together by using two Markov chains for structure completion and a tree search on level slices to guarantee that (1) the combined level had no unbroken structures (e.g., a pipe in *Mario*) and (2) the full level (with a link in the middle of the two level segments if necessary) was completable by an agent. The linking approach was tested on the output of Gram-Elites for the three games, and compared to concatenation. Linking worked for almost every combination of level segments it was given, whereas concatenation often failed for two of the games in the study, but did well on the third.

- Chapter 5 answered **RQ1** by taking the output of Gram-Elites and using the structure, as defined by BCs, to link level segments together to form a digraph. That digraph was used as the structure of an MDP, and a study with agents showed that the MDP could adjust to different player personas for DDA with a simple modification called API, which modified the connections in the MDP by removing actions or edges when players failed too often.

- Chapter 7 answered **RQ2.1** by running a player study to find player ratings of difficulty and enjoyment of level segments for *DungeonGrams*. It was then tested to see what, if any, heuristics could be used to predict the player ratings with a linear regression. The combination of *jaccard-nothing*, *proximity-to-enemies*, *stamina-percent-enemies*, and *density* was most useful for predicting difficulty. The best combination for predicting enjoyment was *path-nothing* and *proximity-to-enemies*.

- Chapter 8 answered **RQ2.2** by using the player ratings gathered in answering **RQ2.1** as rewards for the MDP generated for *DungeonGrams* that was used in Chapter 5, and a player study was run. The best overall condition was the *mean* condition, which assigned every node a reward based on the mean of the difficulty and enjoyment ranking from the player rankings. This chapter also served as an early test to see if the MDP-based approach would work when players interacted with it, and it did work.

- Chapter 9 answered **RQ2** by running a player study for two different games with four different conditions. Two conditions used a new MDP built with *Ponos*, which was a tool built that combined Gram-Elites and linking to output a digraph. The rewards of the two conditions were based on (1) depth and (2) the *mean* of expected difficulty and enjoyment of a given level segment. The third condition built a SLP from the newly generated MDP with *Ponos*. The fourth condition was a handcrafted MDP, and it was built with a new tool called GDM-Editor. The overall best performing condition was the one with the MDP built by hand. However, based on the mPXI, there were almost no differences for player experience.

There were two additional chapters in this work that were not directly related to the research itself. Chapter 6 presented *Ponos*, the name of the software system built that combined Gram-Elites and linking to output a digraph that could be used as the structure for a MDP. Chapter 10 discussed some tips and tricks learned while using *Ponos*.

Answering these research questions led to the Dissertation Statement:

> A Markov Decision Process—which can be created automatically or by hand—can act as a director for DDA via PLG for 2D grid-based games, and the final result is a positive player experience.

Answering **RQ2** gave one definitive answer, which was that the MDP generated by hand resulted in the best player experience. However, that does not mean a better experience could not come from an MDP generated automatically; Chapter 10 offers some thoughts on how this could be accomplished and provides what may be a more promising area which is a hybrid approach, generating parts of the digraph one at a time to build a bigger digraph. But one thing to note here is that the overall method of using an MDP for DDA via PLG did work when tested with agents and, more importantly, when tested with players. The MDP was able to adapt, even when the digraph wasn't perfect—like for the MDP built for *Recformer* in Chapter 9—and give alternative levels for players to play and beat.

11.1 Comparing to Static Level Progressions

This takes us to a topic brought up all the way back in the introduction, which was that designers like to use SLPs for three reasons:

1. *Control* - A SLP lets the designer take full ownership over the player's experience. There are no unexpected outcomes, which is important when the game is a commercial product.

2. *Ease of Implementation* - SLPs are simple to implement. Given a linear progression like *Mario*, you can picture an array of strings, where each string links to a level file. Then the player's current progression is simply an index into that array. Even something more complex, such as a progression with branching, is simply a tree of strings, and a reference to the current node the player is at.

3. *Ease of Modification* - The designer can easily modify the ordering of the levels (e.g., swap strings in a list for a linear progression), remove a level, and add a level. Further, the modification of a level itself is also simple if we assume an adequate level editor.

Overall, whether building an MDP by hand or by using *Ponos*, the designer will have to do more work. That is unavoidable. However, the tradeoff in terms of control, ease of implementation, and ease of modification versus what is gained by using an MDP may be worth it for some designers.

If a designer wants maximum control of the user's experience, they can handcraft the MDP with GDM-Editor. They can set the rewards to try and force the player down their ideal path, but allow for alternatives if the designer's preferred experience is too hard or too difficult.

In terms of ease of implementation, the good news is that the implementation of an MDP for level assembly is already done in both Python and TypeScript. All the designer would have to do is copy and paste the code, and then modify their game to use it. If, however, they want to use another imperative language like C# or C++, the implementation can be done by transpiling the Python or TypeScript code by hand or with a tool. The only major imperative programming language that I am aware of that may cause programmers headaches when converting the code to a different language is Rust, primarily because the way references are used would cause problems with the borrow checker.

Finally, there is an open question about how easy it is to modify an MDP, because no study was run with participants generating MDPs with *Ponos* or making them by hand with GDM-Editor. For *Ponos*, modification is editing a configuration and re-running until satisfied with the result. The problem is that this can be slow and largely based on guesswork until later iterations. For the latter of hand-editing an MDP, modification is simply editing a digraph by hand with GDM-Editor and modifying reward values in a text field. Most modifications are very simple: adding edges, removing edges, adding nodes, and removing nodes. The UI and interactions built for GDM-Editor

were relatively primitive, but they could be improved, and the ease of modification would improve. GDM-Editor could also, with some updates, be used to modify the output of *Ponos*. Overall, the ease of modification is, in my opinion, just slightly worse than a branching SLP.

This leads to one area of future work, which is that GDM-Editor is a relatively primitive tool. I made something that worked for me, but it has rough edges, and those could be smoothed out. A study could be run to find what designers think about the tool, and then improvements could be made based on designer feedback.

11.2 Future Work

There are, of course, many areas for future work. The major ideas have already been listed in each chapter where relevant, but there are two that have not.

Generating an MDP by hand can be just as much work as building a branching progression if the designer only builds one level per node in the digraph. Adding more levels for each node is objectively more work, but it is, I think, worth the time so that if one level is too hard for a player, another level associated with the node may not be. There is, though, a way to potentially reduce the amount of work that the designer has to do by turning GDM-Editor into a co-creative AI tool [72, 104, 106, 128, 190]. Co-creative AI refers to an AI system built to help people with creative tasks.

In this case, the editor could take one level and use it to generate multiple similar levels with n-gram level generation [43], model synthesis [123, 124] / wave-function collapse [93, 95],[1] or constraint solving [34, 35]. Regardless of the method, GDM-Editor would also have to be updated to make server calls to a game server like the one used for *Ponos*, see Chapter 6, to test if the levels generated were completable. The designer, then, could build their progression, and, once they were happy with it, use GDM-Editor to (1) generate new levels for specific nodes and (2) accept or deny the generated levels. The interface could be Tinder-like, with designers swiping to the left to reject a generated level segment or to the right to accept. It could also be improved to include a third option, which allows the designers to edit the generated level themselves before adding it to the node.

The second area for future work is to apply MDP-based level assembly to a different game genre, specifically a *match-3* game.[2] A match-3 game, such as *Bejeweled* [138] and *Candy Crush*

[1] Max Gumin, the creator of Wave Function Collpase, has an implementation in C# available on GitHub: https://github.com/mxgmn/WaveFunctionCollapse

[2] Another type of game that this approach could be applied to is a dungeon-crawler, where levels are not linear (e.g., *Spelunky* [210]).

Figure 11.1: Example screenshot from *Fruit 3*.

Saga [139], is a puzzle game where the player's goal is to manipulate tiles such that three or more tiles line up in a certain shape, such as a line, square, etc. When tiles are matched, they are removed from the screen, and the tiles above fall down to fill in the empty spots. The empty spots at the top are replaced with new tiles. What each new tile will be can be defined probabilistically (i.e., the tile is chosen with weighted random selection) or deterministically (i.e., the tile to fall always follows a certain ordering). The win/loss condition for a match-3 game differs from game to game. A common approach is to say that a player must clear a certain number of tiles of each type in a limited number of moves for the puzzle to be solved.

A game called *Fruit-3*, see Figure 11.1, was made and is playable online.[3] An initial version of a solver was also built for it, along with a server for Gram-Elites; this server was built prior to *Ponos* being built. There are many interesting problems related to how linking would have to be modified to work with a match-3 game to handle resource management. To put it more concretely, the order in which players use resources affects the results. For example, in *DungeonGrams*, grabbing a food too early can make a level impossible to beat. For a match-3 game, the wrong order of a match could make a level impossible to beat if level assembly via stacking level segments on top of each other was used. For a match-3 game, the goal should be to minimize how punishing the linked

[3]Link to playable match-3 game: https://bi3mer.github.io/match-three/

level would be for different orderings; otherwise, winning and losing could feel arbitrary. How this could be done is a topic for future research.

11.3 The End

Thank you all for taking the time to read this dissertation. I hope it was worthwhile and that you consider, if applicable, using an MDP to assemble levels for DDA for your games.

Bibliography

[1] Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1, 2004.

[2] Hervé Abdi. The kendall rank correlation coefficient. *Encyclopedia of Measurement and Statistics. Sage, Thousand Oaks, CA*, pages 508–510, 2007.

[3] Vero Vanden Abeele, Katta Spiel, Lennart Nacke, Daniel Johnson, and Kathrin Gerling. Development and validation of the player experience inventory: A scale to measure player experiences at the level of functional and psychosocial consequences. *International Journal of Human-Computer Studies*, 135:102370, 2020.

[4] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[5] A.I. Design. Rogue, 1980.

[6] Mohammed I. Alhabbash, Ali O. Mahdi, and Samy S. Abu Naser. An intelligent tutoring system for teaching grammar english tenses. *European Academic Research*, 4(9):1–15, 2016.

[7] Maria-Virginia Aponte, Guillaume Levieux, and Stephane Natkin. Measuring the level of difficulty in single player video games. *Entertainment Computing*, 2(4):205–213, 2011.

[8] Richard A Armstrong. When to use the bonferroni correction. *Ophthalmic and physiological optics*, 34(5):502–508, 2014.

[9] Daniel Ashlock and Cameron McGuinness. Automatic generation of fantasy role-playing modules. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8, 2014.

[10] Mahsa Bazzaz and Seth Cooper. Active learning for classifying 2d grid-based level completability. In *2023 IEEE Conference on Games (CoG)*, pages 1–4. IEEE, 2023.

[11] Glen Berseth, M Brandon Haworth, Mubbasir Kapadia, and Petros Faloutsos. Characterizing and optimizing game level difficulty. In *Proceedings of the 7th International Conference on Motion in Games*, pages 153–160, 2014.

[12] Michael Beukman, Christopher W Cleghorn, and Steven James. Procedural content generation using neuroevolution and novelty search for diverse video game levels. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1028–1037, 2022.

[13] Puneeth Bhat, José Moreira, and Satish Kumar Sadasivam. Matrix-multiply assist best practices guide. Technical report, IBM, Tech. Rep., 2021.[Online]. Available: https://www. redbooks. ibm. com . . . , 2021.

[14] Colan Biemer and Seth 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*, pages 1–8, 2024.

[15] Colan Biemer and Seth Cooper. Evaluating the impact of mdp-based level assembly on player experience. In *EXAG@ AIIDE*, 2025.

[16] Colan Biemer, Alejandro Hervella, and Seth Cooper. Gram-elites: N-gram based quality-diversity search. In *Proceedings of the FDG workshop on Procedural Content Generation*, pages 1–6, 2021.

[17] Colan F Biemer and Seth Cooper. On linking level segments. In *2022 IEEE Conference on Games (CoG)*, pages 199–205. IEEE, 2022.

[18] Colan F Biemer and Seth Cooper. Level assembly as a markov decision process. *arXiv preprint arXiv:2304.13922*, 2023.

[19] Carlo Bonferroni. Teoria statistica delle classi e calcolo delle probabilita. *Pubblicazioni del R istituto superiore di scienze economiche e commericiali di firenze*, 8:3–62, 1936.

[20] Jean-Pierre Briot, Gaëtan Hadjeres, and François-David Pachet. Deep learning techniques for music generation–a survey. *arXiv preprint arXiv:1709.01620*, 2017.

[21] Frederick Brooks and H Kugler. *No silver bullet*. April, 1987.

[22] Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[23] Bungie. Halo: Combat evolved, 2001.

[24] Edmund K Burke, Matthew R Hyde, Graham Kendall, and John Woodward. Automatic heuristic generation with genetic programming: evolving a jack-of-all-trades or a master of one. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1559–1565, 2007.

[25] Eric Butler, Erik Andersen, Adam M Smith, Sumit Gulwani, and Zoran Popović. Automatic game progression design through analysis of solution features. In *Proceedings of the 33rd annual ACM conference on human factors in computing systems*, pages 2407–2416, 2015.

[26] Cory J Butz, Shan Hua, and R Brien Maguire. A web-based intelligent tutoring system for computer programming. In *IEEE/WIC/ACM International Conference on Web Intelligence (WI'04)*, pages 159–165. IEEE, 2004.

[27] Szeyi Chan, James Cox, Ala Ebrahimi, Brandon Lyman, and Bob De Schutter. Brukel vs brukel: Impact of game fidelity on player experience in gaminiscing games. In *2023 IEEE Conference on Games (CoG)*, pages 1–4. IEEE, 2023.

[28] S.F. Chen and R. Rosenfeld. A survey of smoothing techniques for me models. *IEEE Transactions on Speech and Audio Processing*, 8(1):37–50, 2000.

[29] Israel Cohen, Yiteng Huang, Jingdong Chen, Jacob Benesty, Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. *Noise reduction in speech processing*, pages 1–4, 2009.

[30] Steve Colley and Howard Palmer. Maze war, 1973.

[31] Kate Compton and Michael Mateas. Casual creators. In *ICCC*, pages 228–235, 2015.

[32] Kate Compton and Michael Mateas. Casual creators. In *ICCC*, pages 228–235, 2015.

[33] Michael Cook, Simon Colton, and Jeremy Gow. The ANGELINA videogame design system—part I. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(2):192–203, 2016.

[34] Seth Cooper. Sturgeon: tile-based procedural level generation via learned and designed constraints. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 18, pages 26–36, 2022.

[35] Seth Cooper. Sturgeon-graph: Constrained graph generation from examples. In *Proceedings of the 18th International Conference on the Foundations of Digital Games*, pages 1–9, 2023.

[36] Seth Cooper and Mahsa Bazzaz. Literally unplayable: On constraint-based generation of uncompletable levels. In *Proceedings of the 19th International Conference on the Foundations of Digital Games*, pages 1–8, 2024.

[37] Seth Cooper and Anurag Sarkar. Pathfinding agents for platformer level repair. In *AIIDE Workshops*, 2020.

[38] Albert T Corbett, Kenneth R Koedinger, and John R Anderson. Intelligent tutoring systems. In *Handbook of human-computer interaction*, pages 849–874. Elsevier, 1997.

[39] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[40] Brent Cowan and Bill Kapralos. A simplified level editor. In *2011 IEEE International Games Innovation Conference (IGIC)*, pages 52–54. IEEE, 2011.

[41] Antoine Cully, Jeff Clune, Danesh Tarapore, and Jean-Baptiste Mouret. Robots that can adapt like animals. *Nature*, 521(7553):503–507, 2015.

[42] Mihaly Czikszentmihalyi. *Flow: The psychology of optimal experience*. New York: Harper & Row, 1990.

[43] Steve Dahlskog, Julian Togelius, and Mark J Nelson. Linear levels through n-grams. In *Proceedings of the 18th International Academic MindTrek Conference: Media Business, Management, Content & Services*, pages 200–206, 2014.

[44] Cecilia D'Anastasio. Why 2021 was the biggest year for the labor movement in games, Dec 2021.

[45] Charles Darwin, John Wyon Burrow, and John Wyon Burrow. *The origin of species by means of natural selection: or, the preservation of favored races in the struggle for life*. AL Burt New York, 2009.

[46] Florent Delgrange, Joost-Pieter Katoen, Tim Quatmann, and Mickael Randour. Simple strategies in multi-objective mdps. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 346–364. Springer, 2020.

[47] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186, 2019.

[48] Alexis Dinno. Nonparametric pairwise multiple comparisons in independent groups using dunn's test. *The Stata Journal*, 15(1):292–300, 2015.

[49] Emily Dolson, Alexander Lalejini, and Charles Ofria. Exploring genetic programming systems with map-elites. *Genetic Programming Theory and Practice XVI*, pages 1–16, 2019.

[50] Joris Dormans. Adventures in level design: generating missions and spaces for action adventure games. In *Proceedings of the 2010 workshop on procedural content generation in games*, pages 1–8, 2010.

[51] Jimmy Efird. Blocked randomization with randomly selected block sizes. *International journal of environmental research and public health*, 8(1):15–20, 2011.

[52] Kousha Etessami, Marta Kwiatkowska, Moshe Y Vardi, and Mihalis Yannakakis. Multi-objective model checking of markov decision processes. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 50–65. Springer, 2007.

[53] Silvia Ferrari and Francisco Cribari-Neto. Beta regression for modelling rates and proportions. *Journal of applied statistics*, 31(7):799–815, 2004.

[54] Matthew C. Fontaine, Ruilin Liu, Ahmed Khalifa, Jignesh Modi, Julian Togelius, Amy K. Hoover, and Stefanos Nikolaidis. Illuminating Mario scenes in the latent space of a Generative Adversarial Network. *arXiv:2007.05674 [cs]*, December 2020.

[55] From Software inc. Elden ring, 2022.

[56] Ojiro Fumoto. Downwell, 2015.

[57] Joseph L Gastwirth, Yulia R Gel, and Weiwen Miao. The impact of levene's test of equality of variances on statistical theory and practice. 2009.

[58] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo= asp+ control: Preliminary report. *arXiv preprint arXiv:1405.3694*, 2014.

[59] Alexander Gellel and Penny Sweetser. A hybrid approach to procedural generation of roguelike video game levels. In *Proceedings of the 15th International Conference on the Foundations of Digital Games*, pages 1–10, 2020.

[60] Kathrin M Gerling, Max Birk, Regan L Mandryk, and Andre Doucette. The effects of graphical fidelity on player experience. In *Proceedings of international conference on Making Sense of Converging Media*, pages 229–236, 2013.

[61] Mark E Glickman. Example of the glicko-2 system. *Boston University*, 28, 2012.

[62] Miguel González-Duque, Rasmus Berg Palm, David Ha, and Sebastian Risi. Finding game levels with the right difficulty in a few trials through intelligent trial-and-error. In *2020 IEEE Conference on Games (CoG)*, pages 503–510. IEEE, 2020.

[63] Miguel Gonzalez-Duque, Rasmus Berg Palm, and Sebastian Risi. Fast game content adaptation through bayesian-based player modelling. In *2021 IEEE Conference on Games (CoG)*, pages 01–08. IEEE, 2021.

[64] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[65] Arthur C Graesser, Mark W Conley, and Andrew Olney. Intelligent tutoring systems. 2012.

[66] Jens Gramm, Jiong Guo, Falk Hüffner, Rolf Niedermeier, Hans-Peter Piepho, and Ramona Schmid. Algorithms for compact letter displays: Comparison and evaluation. *Computational Statistics & Data Analysis*, 52(2):725–736, 2007.

[67] Daniele Gravina, Ahmed Khalifa, Antonios Liapis, Julian Togelius, and Georgios N Yannakakis. Procedural content generation through quality diversity. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2019.

[68] Michael Cerny Green, Ahmed Khalifa, Gabriella AB Barros, Andy Nealen, and Julian Togelius. Generating levels that teach mechanics. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, pages 1–8, 2018.

[69] Michael Cerny Green, Luvneesh Mugrai, Ahmed Khalifa, and Julian Togelius. Mario level generation from mechanics using scene stitching. In *2020 IEEE Conference on Games (CoG)*, pages 49–56. IEEE, 2020.

[70] Matej Guid and Ivan Bratko. Search-based estimation of problem difficulty for humans. In *Artificial Intelligence in Education: 16th International Conference, AIED 2013, Memphis, TN, USA, July 9-13, 2013. Proceedings 16*, pages 860–863. Springer, 2013.

[71] Matthew Guzdial, Nicholas Liao, Jonathan Chen, Shao-Yu Chen, Shukan Shah, Vishwa Shah, Joshua Reno, Gillian Smith, and Mark 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*, pages 1–13, 2019.

[72] Matthew Guzdial and Mark Riedl. An interaction framework for studying co-creative ai. *arXiv preprint arXiv:1903.09709*, 2019.

[73] Matthew Guzdial, Sam Snodgrass, and Adam J Summerville. *Procedural content generation via machine learning: An overview*. Springer, 2022.

[74] Aqeel Haider, Casper Harteveld, Daniel Johnson, Max V Birk, Regan L Mandryk, Magy Seif El-Nasr, Lennart E Nacke, Kathrin Gerling, and Vero 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(CHI PLAY):1–26, 2022.

[75] Nikolaus Hansen, Sibylle D Müller, and Petros Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evolutionary computation*, 11(1):1–18, 2003.

[76] Karin Harbusch, Gergana Itsova, Ulrich Koch, and Christine Kühner. The sentence fairy: a natural-language generation system to support children's essay writing. *Computer Assisted Language Learning*, 21(4):339–352, 2008.

[77] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[78] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[79] Jacob Heller. *Catch-22*. Simon & Schuster, 1961.

[80] Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, pages 65–70, 1979.

[81] Danial Hooshyar, Moslem Yousefi, and Heuiseok Lim. Data-driven approaches to game player modeling: a systematic literature review. *ACM Computing Surveys (CSUR)*, 50(6):1–19, 2018.

[82] Dayana Hristova, Matej Guid, and Ivan Bratko. Assessing the difficulty of chess tactical problems. *International Journal on Advances in Intelligent Systems*, 7(3):728–738, 2014.

[83] Tobias Huber, Silvan Mertes, Stanislava Rangelova, Simon Flutura, and Elisabeth André. Dynamic difficulty adjustment in virtual reality exergames through experience-driven procedural content generation. In *2021 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8. IEEE, 2021.

[84] Robin Hunicke. The case for dynamic difficulty adjustment in games. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 429–433, 2005.

[85] Erdi Izgi. Framework for roguelike video games development. 2018.

[86] Paul Jaccard. Nouvelles recherches sur la distribution florale. *Bull. Soc. Vaud. Sci. Nat.*, 44:223–270, 1908.

[87] Rishabh Jain, Aaron Isaksen, Christoffer Holmgård, and Julian Togelius. Autoencoders for level generation, repair, and recognition. In *Proceedings of the ICCC workshop on computational creativity and games*, volume 9, 2016.

[88] Chaima Jemmali, Carter Ithier, Seth Cooper, and Magy Seif El-Nasr. Grammar based modular level generator for a programming puzzle game. In *AIIDE Workshop*, 2020.

[89] Chaima Jemmali, Magy Seif El-Nasr, and Seth Cooper. The effects of adaptive procedural levels on engagement and performance in an educational programming game. In *Proceedings of the 17th International Conference on the Foundations of Digital Games*, pages 1–12, 2022.

[90] Martin Jennings-Teats, Gillian Smith, and Noah Wardrip-Fruin. Polymorph: dynamic difficulty adjustment through level generation. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pages 1–4, 2010.

[91] Dan Jurafsky. *Speech & language processing*. Pearson Education India, 2000.

[92] Minsoo Kang, Brian G Ragan, and Jae-Hyeon Park. Issues in outcomes research: an overview of randomization techniques for clinical trials. *Journal of athletic training*, 43(2):215–221, 2008.

[93] Isaac Karth and Adam M Smith. Wavefunctioncollapse is constraint solving in the wild. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, pages 1–10, 2017.

[94] Isaac Karth and Adam M Smith. Addressing the fundamental tension of pcgml with discriminative learning. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, pages 1–9, 2019.

[95] Isaac Karth and Adam M Smith. Wavefunctioncollapse: Content generation via constraint solving and machine learning. *IEEE Transactions on Games*, 14(3):364–376, 2021.

[96] Kalev Kask and Rina Dechter. A general scheme for automatic generation of search heuristics from specification dependencies. *Artificial Intelligence*, 129(1-2):91–131, 2001.

[97] S. Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35(3):400–401, 1987.

[98] Walter N Kernan, Catherine M Viscoli, Robert W Makuch, Lawrence M Brass, and Ralph I Horwitz. Stratified randomization for clinical trials. *Journal of clinical epidemiology*, 52(1):19–26, 1999.

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

[100] Ahmed Khalifa, Michael Cerny Green, Gabriella Barros, and Julian Togelius. Intentional computational level design. In *Proceedings of The Genetic and Evolutionary Computation Conference*, pages 796–803, 2019.

[101] Ahmed Khalifa, Scott Lee, Andy Nealen, and Julian Togelius. Talakat: Bullet hell generation through constrained map-elites. In *Proceedings of The Genetic and Evolutionary Computation Conference*, pages 1047–1054, 2018.

[102] Steven Orla Kimbrough, Gary J Koehler, Ming Lu, and David Harlan Wood. Introducing a feasible-infeasible two-population (fi-2pop) genetic algorithm for constrained optimization: Distance tracing and no free lunch. *European Journal of Operational Research*, 2005.

[103] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[104] Max Kreminski, Melanie Dickinson, Michael Mateas, and Noah Wardrip-Fruin. Why are we like this?: The ai architecture of a co-creative storytelling game. In *Proceedings of the 15th International Conference on the Foundations of Digital Games*, pages 1–4, 2020.

[105] Francois Dominic Laramee. Using n-gram statistical models to predict player behavior. *AI Game Programming Wisdom, Charles River Media, Inc., Hingham, MA., USA*, pages 596–601, 2002.

[106] Tinea Larsson, Jose Font, and Alberto Alvarez. Towards ai as a creative colleague in game level design. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 18, pages 137–145, 2022.

[107] Joel Lehman and Kenneth O Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 19(2):189–223, 2011.

[108] Bowei Li, Ruohan Chen, Yuqing Xue, Ricky Wang, Wenwen Li, and Matthew Guzdial. Ensemble learning for mega man level generation. *arXiv preprint arXiv:2107.12524*, 2021.

[109] Antonios Liapis. Multi-segment evolution of dungeon game levels. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 203–210, 2017.

[110] Antonios Liapis, Christoffer Holmgård, Georgios N Yannakakis, and Julian Togelius. Procedural personas as critics for dungeon generation. In *European Conference on the Applications of Evolutionary Computation*, pages 331–343. Springer, 2015.

[111] Yunzhi Lin, Ming Zhu, and Zheng Su. The pursuit of balance: an overview of covariate-adaptive randomization techniques in clinical trials. *Contemporary clinical trials*, 45:21–25, 2015.

[112] Conor Linehan, George Bellord, Ben Kirman, Zachary H Morford, and Bryan Roche. Learning curves: analysing pace and challenge in four successful puzzle games. In *Proceedings of the first ACM SIGCHI annual symposium on Computer-human interaction in play*, pages 181–190, 2014.

[113] Michael L Littman. A tutorial on partially observable markov decision processes. *Journal of Mathematical Psychology*, 53(3):119–125, 2009.

[114] ManYat Lo and Simon M Lucas. Evolving musical sequences with n-gram based trainable fitness functions. In *2006 IEEE International Conference on Evolutionary Computation*, pages 601–608. IEEE, 2006.

[115] Brian Magerko, John E Laird, Mazin Assanie, Alex Kerfoot, Devvan Stokes, et al. Ai characters and directors for interactive computer games. In *AAAI*, pages 877–883, 2004.

[116] Travis Mandel, Yun-En Liu, Sergey Levine, Emma Brunskill, and Zoran Popovic. Offline policy evaluation across representations with applications to educational games. In *AAMAS*, volume 1077, 2014.

[117] Julian Mariño, Willian Reis, and Levi Lelis. An empirical evaluation of evaluation metrics of procedurally generated mario levels. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 11, pages 44–50, 2015.

[118] Peyman Massoudi and Amir H Fassihi. Achieving dynamic ai difficulty by using reinforcement learning and fuzzy logic skill metering. In *IGIC*, pages 163–168, 2013.

[119] Patrick E McKight and Julius Najab. Kruskal-wallis test. *The corsini encyclopedia of psychology*, pages 1–1, 2010.

[120] Ryan P McMahan, Doug A Bowman, David J Zielinski, and Rachael B Brady. Evaluating display fidelity and interaction fidelity in a virtual reality game. *IEEE transactions on visualization and computer graphics*, 18(4):626–633, 2012.

[121] Edmund McMillen, Florian Himsl, and Danny Baranowsky. Binding of isaac, 2011.

[122] Danielle S McNamara, G Tanner Jackson, and Art Graesser. Intelligent tutoring and games (itag). In *Gaming for classroom-based learning: Digital role playing as a motivator of study*, pages 44–65. IGI Global, 2010.

[123] Paul Merrell. Example-based model synthesis. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 105–112, 2007.

[124] Paul Merrell and Dinesh Manocha. Model synthesis: A general procedural modeling algorithm. *IEEE transactions on visualization and computer graphics*, 17(6):715–728, 2010.

[125] Ian Millington. *AI for Games*. CRC Press, 2019.

[126] Christopher M Mitchell, Kristy Elizabeth Boyer, and James C Lester. A markov decision process model of tutorial intervention in task-oriented dialogue. In *Artificial Intelligence in Education: 16th International Conference, AIED 2013, Memphis, TN, USA, July 9-13, 2013. Proceedings 16*, pages 828–831. Springer, 2013.

[127] Mojang Studios. Minecraft, 2011.

[128] Caterina Moruzzi and Solange Margarido. A user-centered framework for human-ai cocreativity. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, pages 1–9, 2024.

[129] Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

[130] Naughty Dog. Crash bandicoot, 1996.

[131] Nintendo. Super mario bros., 1985.

[132] Nintendo. Kid icarus, 1986.

[133] Scott Novotney and Chris Callison-Burch. Crowdsourced accessibility: Elicitation of wikipedia articles. In *Proceedings of the NAACL HLT 2010 workshop on creating speech and language data with Amazon's Mechanical Turk*, pages 41–44, 2010.

[134] Number None. Braid, 2008.

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

[136] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: an analysis and survey. In *Advances in computers*, volume 112, pages 275–378. Elsevier, 2019.

[137] Ken Perlin. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):287–296, 1985.

[138] PopCap Games. Bejeweled, 2001.

[139] PopCap Games. Candy crush saga, 2012.

[140] Justin K Pugh, Lisa B Soros, and Kenneth O Stanley. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI*, 3:202845, 2016.

[141] Willian M. P. Reis, Levi H. S. Lelis, and Ya'akov Kobi Gal. Human computation for procedural content generation in platform games. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 99–106, 2015.

[142] Charles Reynaldo, Ryan Christian, Hansel Hosea, and Alexander AS Gunawan. Using video games to improve capabilities in decision making and cognitive skill: A literature review. *Procedia Computer Science*, 179:211–221, 2021.

[143] Scott Rogers. *Level Up! The guide to great video game design*. John Wiley & Sons, 2014.

[144] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[145] Stuart Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Pearson, Upper Saddle River, 3rd edition edition, December 2009.

[146] Christoph Salge, Michael Cerny Green, Rodgrigo Canaan, and Julian Togelius. Generative design in minecraft (gdmc) settlement generation competition. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, pages 1–10, 2018.

[147] Christoph Salge, Christian Guckelsberger, Michael Cerny Green, Rodrigo Canaan, and Julian Togelius. Generative design in minecraft: Chronicle challenge. *arXiv preprint arXiv:1905.05888*, 2019.

[148] Pedro Sampaio, Augusto Baffa, Bruno Feijó, and Mauricio Lana. A fast approach for automatic generation of populated maps with seed and difficulty control. In *2017 16th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, pages 10–18. IEEE, 2017.

[149] Anurag Sarkar and Seth Cooper. Blending levels from different games using lstms. In *AIIDE Workshops*, 2018.

[150] Anurag Sarkar and Seth Cooper. Transforming game difficulty curves using function composition. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–7, 2019.

[151] Anurag Sarkar and Seth Cooper. Sequential segment-based level generation and blending using variational autoencoders. In *International Conference on the Foundations of Digital Games*, pages 1–9, 2020.

[152] Noor Shaker, Miguel Nicolau, Georgios N Yannakakis, Julian Togelius, and Michael O'neill. Evolving levels for super mario bros using grammatical evolution. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 304–311. IEEE, 2012.

[153] Noor Shaker, Julian Togelius, and Mark J Nelson. *Procedural content generation in games*. Springer, 2016.

[154] Noor Shaker, Julian Togelius, Georgios N Yannakakis, Ben Weber, Tomoyuki Shimizu, Tomonori Hashiyama, Nathan Sorenson, Philippe Pasquier, Peter Mawhorter, Glen Takahashi, et al. The 2010 mario ai championship: Level generation track. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(4):332–347, 2011.

[155] Samuel Sanford Shapiro and Martin B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4):591–611, 1965.

[156] Evan C Sheffield and Michael D Shah. Dungeon digger: Apprenticeship learning for procedural dungeon building agents. In *Proceedings of the 2018 Annual Symposium on Computer-Human Interaction in Play Companion Extended Abstracts*, pages 603–610, 2018.

[157] Shu-Chuan Shih, Chih-Chia Chang, Bor-Chen Kuo, and Yu-Han Huang. Mathematics intelligent tutoring system for learning multiplication and division of fractions based on diagnostic teaching. *Education and Information Technologies*, 28(7):9189–9210, 2023.

[158] Tanya Short and Tarn Adams. *Procedural generation in game design*. CRC Press, 2017.

[159] Tianye Shu, Jialin Liu, and Georgios N Yannakakis. Experience-driven pcg via reinforcement learning: A super mario bros study. In *2021 IEEE Conference on Games (CoG)*, pages 1–9. IEEE, 2021.

[160] Adam M Smith, Eric Butler, and Zoran Popovic. Quantifying over play: Constraining undesirable solutions in puzzle design. In *FDG*, pages 221–228, 2013.

[161] Gillian Smith, Mike Treanor, Jim Whitehead, and Michael Mateas. Rhythm-based level generation for 2d platformers. In *Proceedings of the 4th international Conference on Foundations of Digital Games*, pages 175–182, 2009.

[162] Gillian Smith and Jim Whitehead. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pages 1–7, 2010.

[163] Sam Snodgrass and Santiago Ontanon. A hierarchical approach to generating maps using markov chains. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 10, 2014.

[164] Sam Snodgrass and Santiago Ontanón. Controllable procedural content generation via constrained multi-dimensional markov chain sampling. In *IJCAI*, pages 780–786, 2016.

[165] Sam Snodgrass and Santiago Ontanón. Procedural level generation using multi-layer level representations with mdmcs. In *2017 IEEE conference on computational intelligence and games (CIG)*, pages 280–287. IEEE, 2017.

[166] Edward Jay Sondik. *The optimal control of partially observable Markov processes*. Stanford University, 1971.

[167] Steven Spielberg. Raiders of the lost ark, 1981.

[168] Frank Spitzer. *Principles of random walk*, volume 34. Springer Science & Business Media, 2001.

[169] Lars St, Svante Wold, et al. Analysis of variance (anova). *Chemometrics and intelligent laboratory systems*, 6(4):259–272, 1989.

[170] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[171] Gail M Sullivan and Anthony R Artino Jr. Analyzing and interpreting data from likert-type scales. *Journal of graduate medical education*, 5(4):541–542, 2013.

[172] Adam Summerville. Expanding expressive range: Evaluation methodologies for procedural content generation. In *Fourteenth artificial intelligence and interactive digital entertainment conference*, 2018.

[173] Adam Summerville and Michael Mateas. Super mario as a string: Platformer level generation via lstms. *arXiv preprint arXiv:1603.00930*, 2016.

[174] Adam Summerville, Shweta Philip, and Michael Mateas. Mcmcts pcg 4 smb: Monte carlo tree search to guide platformer level generation. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 11, 2015.

[175] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. Procedural content generation via machine learning (pcgml). *IEEE Transactions on Games*, 10(3):257–270, 2018.

[176] Adam James Summerville, Sam Snodgrass, Michael Mateas, and Santiago Ontanón. The vglc: The video game level corpus. *arXiv preprint arXiv:1606.07487*, 2016.

[177] SuperGiant Games. Hades, 2020.

[178] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[179] Mandy JW Tak, Mark HM Winands, and Yngvi Bjornsson. N-grams and the last-good-reply policy applied in general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):73–83, 2012.

[180] Ryo Takaoka, Masayuki Shimokawa, and Toshio Okamoto. A framework of educational control in game-based learning environment. In *2011 IEEE 11th International Conference on Advanced Learning Technologies*, pages 32–36, 2011.

[181] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[182] Team Meat. Super meat boy, 2010.

[183] David Thue and Vadim Bulitko. Procedural game adaptation: Framing experience management as changing an MDP. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 8(2):44–50, 2012.

[184] Julian Togelius, Sergey Karakovskiy, and Robin Baumgarten. The 2009 mario ai competition. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2010.

[185] Julian Togelius, Sergey Karakovskiy, and Robin Baumgarten. The 2009 mario ai competition. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[186] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011.

[187] Jonathan Tremblay, Pedro Andrade Torres, and Clark Verbrugge. Measuring risk in stealth games. In *FDG*. Citeseer, 2014.

[188] John W Tukey. Comparing individual means in the analysis of variance. *Biometrics*, pages 99–114, 1949.

[189] Anant J Umbarkar and Pranali D Sheth. Crossover operators in genetic algorithms: a review. *ICTACT journal on soft computing*, 6(1), 2015.

[190] Josh Urban Davis, Fraser Anderson, Merten Stroetzel, Tovi Grossman, and George Fitzmaurice. Designing co-creative ai for virtual environments. In *Proceedings of the 13th Conference on Creativity and Cognition*, pages 1–11, 2021.

[191] Valve Corporation. Left 4 dead 2, 2009.

[192] Roland Van der Linden, Ricardo Lopes, and Rafael Bidarra. Designing procedurally generated levels. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[193] Marc van Kreveld, Maarten Löffler, and Paul Mutser. Automated puzzle difficulty estimation. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 415–422, 2015.

[194] Joseph II Vasquez. Implementing n-grams for player prediction, procedural generation, and stylized ai. *Game AI Pro*, pages 567–580, 2014.

[195] Estela Aparecida Oliveira Vieira, Aleph Campos da SILVEIRA, and Ronei Ximenes Martins. Heuristic evaluation on usability of educational games: A systematic review. *Informatics in Education*, 18(2):427–442, 2019.

[196] Vanessa Volz, Jacob Schrum, Jialin Liu, Simon M Lucas, Adam Smith, and Sebastian Risi. Evolving mario levels in the latent space of a deep convolutional generative adversarial network. In *Proceedings of the genetic and evolutionary computation conference*, pages 221–228, 2018.

[197] Carl Vondrick, Hamed Pirsiavash, and Antonio Torralba. Generating videos with scene dynamics. *Advances in neural information processing systems*, 29, 2016.

[198] Hao Wang, Yu-Wen Wang, and Chuen-Tsai Sun. Rating logic puzzle difficulty automatically in a human perspective. In *Proceedings of Nordic DiGRA 2012 Conference*, 2012.

[199] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[200] Michael Weeks and Jonathon Davis. Procedural dungeon generation for a 2d top-down game. In *Proceedings of the 2022 ACM Southeast Conference*, pages 60–66, 2022.

[201] R Clinton Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 38–38. IEEE, 1998.

[202] Sandra Williams. Generating mathematical word problems. In *AAAI Fall Symposium: Question Generation*, 2011.

[203] Oliver Withington. Illuminating super mario bros: quality-diversity within platformer level generation. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 223–224, 2020.

[204] Oliver Withington, Michael Cook, and Laurissa Tokarchuk. On the evaluation of procedural level generation systems. *arXiv preprint arXiv:2404.18657*, 2024.

[205] Tai-Hsi Wu, Shu-Hsing Chung, and Chin-Chih Chang. Hybrid simulated annealing algorithm with mutation operator to the cell formation problem with alternative process routings. *Expert Systems with Applications*, 36(2):3652–3661, 2009.

[206] Su Xue, Meng Wu, John Kolen, Navid Aghdaie, and Kazi A Zaman. Dynamic difficulty adjustment for maximized engagement in digital games. In *Proceedings of the 26th International Conference on World Wide Web Companion*, pages 465–471, 2017.

[207] Geogios N Yannakakis. Game ai revisited. In *Proceedings of the 9th conference on Computing Frontiers*, pages 285–292, 2012.

[208] Georgios N. Yannakakis, Pieter Spronck, Daniele Loiacono, and Elisabeth André. Player modeling. In *Artificial and Computational Intelligence in Games*, 2013.

[209] Georgios N Yannakakis and Julian Togelius. *Artificial intelligence and games*, volume 2. Springer, 2018.

[210] Derek Yu. *Spelunky: Boss Fight Books# 11*, volume 11. Boss Fight Books, 2016.

[211] Kristen Yu, Matthew Guzdial, Nathan R Sturtevant, Morgan Cselinacz, Chris Corfe, Izzy Hubert Lyall, and Chris Smith. Adventures of ai directors early in the development of nightingale. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 18, pages 70–77, 2022.

[212] Jerrold H Zar. Spearman rank correlation. *Encyclopedia of biostatistics*, 7, 2005.

[213] Hejia Zhang, Matthew Fontaine, Amy Hoover, Julian Togelius, Bistra Dilkina, and Stefanos Nikolaidis. Video game level repair via mixed integer linear programming. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 16(1):151–158, October 2020. Number: 1.

[214] Mohammad Zohaib. Dynamic difficulty adjustment (dda) in computer games: A review. *Advances in Human-Computer Interaction*, 2018, 2018.

Index