Abstract

This project explores how to best generate puzzle games that require path planning to solve. In particular, its focus is narrowed to a subset of puzzle games that challenge the player to find a path from point A to point B. This path may not be readily available to the player and must be found by them activating a series of “switches.” These “switches” modify the game environment and, consequently, the paths available to navigate. The goal of this project is to develop an algorithm which can automatically generate this type of puzzle game. To achieve this, we looked to a specific path-puzzle game, called Monument Valley, for a basis to begin experimentation. Specifically, the work for the project required Unity to build Monument Valley styled puzzle game levels and C# to program the “switches” and conduct the level generation.
Acknowledgments

Thank you to my thesis advisor, Aline Normoyle for all the help and guidance given over the course of this semester. And to my friends and family for offering support and advice when needed.
## Contents

1. Introduction ................................................................. 7

2. Related Work ............................................................... 8
   2.1 Grammar Based PCG .................................................. 8
   2.2 Answer Set Programming Based PCG ............................. 9
   2.3 Wave Function Collapse Based PCG .............................. 10
   2.4 Evolutionary Search Based PCG ................................. 11
   2.5 Test-and-Reject Based PCG ....................................... 13
   2.6 Influences on Our Implementation ............................. 14

3. Methodology/Efforts To Date ........................................ 14
1 Introduction

Procedural content generation (PCG) is the use of algorithms to produce game content automatically [TSN16]. This content can range from the game rules, characters, goals, and scenery needed for the game design. A particularly fruitful application of PCG is the generation of game levels. Such generation enables games to have the potential for an infinite number of levels, preventing users from ever running out of levels to complete. Because of this appeal, a lot of research and work have gone into this topic, resulting in a variety of algorithms aimed to produce successful game levels efficiently [TSN16]. These algorithms include grammars [TSD16], answer set programming [NS16], wave-function collapse [KS17], evolutionary search [TS16], and test-and-reject [DB17]. Each of these derives its basis from different fields and backgrounds but has been proven to function well for procedural level generation.

In addition to various types of PCG algorithms, there are also a variety of game types that look to benefit from procedural level generation. From First Person Shooter games [TS16] to dungeon set games [SLT16] to puzzle games [KSG16], each application contains its own special set of problems when attempting to generate playable levels. The work for this project analyzes how procedural level generation can best be implemented for Monument Valley styled puzzle games, see Figure 1 for a visual reference of what these levels look like in the commercial version of Monument Valley. Note that in Figure 1 the goal is to find a path from the orange character in the bottom, point A, to the black square-shaped target in the top, point B. Monument Valley was awarded the Apple Design Award in 2014 and has earned high praise and reviews for its compelling puzzles and stunning visuals [Tac14]. It is, however, rather short in number of playable levels [Rie14]. PCG’s potential to infinitely expand on this number of playable levels make Monument Valley a great candidate for the work of this project. Focusing on the Monument Valley style of puzzle games involves a new set of problems for the level generation beyond those normally involved in level generation for other puzzle games. This is due to Monument Valley’s unique use of “switches” to edit the game space.

In this style of puzzle games, the path from point A to point B may not be readily available to the player and must be found by activating a series of “switches.” These “switches” modify the game environment and, consequently, the paths available to navigate. Monument Valley portrays the game environment from an isometric perspective. Its use of an isometric perspective lets the “switches” occur in unexpected ways.

Figure 1: Monument Valley Reference Image.
The player can unlock paths through Escher-like illusions to reach the level’s goal position.

The inclusion of such “switches” within the game impacts how the level generation can occur. Special consideration must now be given to how to abstractly represent the environment, including the “switches”, and which style of PCG algorithm will produce playable, fun levels of this kind efficiently. The result of this project is a rule based generation algorithm. With it, one can produce playable puzzle game levels of the *Monument Valley* style. To the best of our knowledge, this is the first system that can procedurally generate game levels of this kind. This paper discusses existing implementations of various PCG algorithms for level generation with puzzle games, then goes on to outline the approaches and results of our use of PCG for the *Monument Valley* style game levels.

## 2 Related Work

The field of Procedural Content Generation has garnered a large influx of research in recent years [TSN16]. Many algorithms that originated in fields like linguistics [TSD16] and artificial intelligence [ART16] have found applications within PCG, and specifically for procedural level generation. The primary aspects of PCG for level generation involve content representation and actual content creation [TSN16]. Each algorithm has a particular way of achieving these two aspects, as well as focusing on other specific aspects which they attempt to optimize or control. We see this in the following analyses of some of the major types of PCG algorithms for level generation.

### 2.1 Grammar Based PCG

Grammar based PCG is heavily influenced by linguistic and natural language based work. Initially, grammars were simply sets of rules for rewriting strings [TSD16]. At the base level, this is essentially what they do within PCG as well. In the PCG implementation of grammars, sets of rules are used to expand strings of symbols that represent commands like ‘F’ for forward, ‘+’ for right, ‘-’ for left. Additionally, grammars can be bracketed. This allows for a grammar to return to a previously saved position after performing a series of actions on a later position in the string [TSD16].

Grammars can also be used to produce graphs, tile maps, shapes, etc. as well as the string expansion explained above [TSD16]. Such graph generation makes them useful for generating game spaces, especially if the game space allows for multiple paths to the level’s goal position. Similar to the way they expand strings, grammars use sets of rules to construct their graphs. These rules can be thought of as steps within the design process for the level and its graph representation. Additionally, grammar based PCG for levels often breaks up the generation into two sub-tasks: game space generation and level “mission” generation. Game space generation involves defining the shape/layout of the level and what is included in the game environment. Level “mission” generation involves determining the goal for the player solving that level. Generating these separately allows for different styles of representation and rule sets to be used. Linking them back together, in the end, produces a completely generated level [TSD16].

Grammars have a wide range of applications within PCG. Jemmali et al. used it to
procedurally generate levels for an education focused puzzle game, May’s Journey \cite{ART16}. The authors let the grammar based method handle the content generation aspects within the level generation while implementing a “work backward” approach to ensure the generated levels are solvable without needing post-processing or filtering. Specifically, the authors converted the input, a random solution, to an abstract syntax tree. From the tree, the game object that should be involved in the level is easily extracted, as well as the actions and attributes ascribed to it. This is passed through the grammar that used a series of rules to determine what shape the game environment should take to satisfy all the requirements. The rules within the set were weighted with probabilities to favor certain rules over others. Additionally, penalties were implemented when rules were selected so that no one rule was disproportionally chosen. Each object specific map shape was combined to a final game space by minimizing the number of modifications needed for such merging. Figure 2 shows two object specific map shapes combined to a full map in both the abstract representation, left, and the actual game representation, right. Finally, either a reward path algorithm or a maze path algorithm was used to determine the path through the level. Either algorithm maintained entrance-to-exit navigability \cite{ART16}.

The results of the grammar based PCG approach efficiently produced solvable levels with a high amount of variation. The paper identified, however, that the method required a fair amount of familiarity with the game’s reliance on coding conventions. A similar application of this method for our Monument Valley styled game levels presented the opportunity to make use of the highly successful model. We could avoid this identified dependence on coding conventions as the gameplay within Monument Valley styled games consists of fairly simple point-and-click environment navigation.

However, from a large-scale perspective, grammars can be difficult to implement as they require a deep understanding of the domain being modeled. Intricacies in the Monument Valley style levels may make this task especially complex. Due to such constraints, a less design-reliant approach may be more applicable for our generator.

### 2.2 Answer Set Programming Based PCG

Answer set programming (ASP) is a logic programming approach for PCG \cite{NS16}. ASP is used to specify what the generated content should be like and then an ASP solver produces the content from this program. The first step in this process is defining the game logic i.e. the game structure and mechanics. Additionally, the constraints for how the content
should adhere to certain properties must be defined. Once these are outlined for the ASP, the ASP solver can produce and find content that matches such input. ASP is a well-defined and frequently used method for PCG. This is largely due to the existing reliable tools like AnsProlog and Clingo that take the input required for ASP and efficiently produce the valid output [NS16].

One specific use of ASP was outlined by Lindeman to generate levels for a self-created puzzle game called Swappy [Lin18]. See Figure 3 for an example of the Swappy game interface. In this application of ASP, the method of “core-sim-style” [Lin18] was implemented in which three AnsProlog programs were developed. One, core, was used to generate the game pieces and explain the relationship between them. The second, sim, acted as a simulation of gameplay to ensure the rules of gameplay are followed. The final, style, was used to generate artifacts with a mind towards some aesthetic criteria. These three programs were linked by a Python command-line tool that took in parameters defining the level width and the number of character tokens to be used in the level. The AnsProlog files were run by the Clingo ASP solver. Of the many output levels, one was randomly selected once each ASP program had run. The Python tool then parsed and rendered the facts of the answer set chosen into ASCII art in the terminal. This output was finally provided to the Swappy game client [Lin18].

The results of this implementation of ASP were mixed. Lindeman noted that the generator occasionally still produced unsolvable levels. This was largely due to the bugs surrounding how the win condition of the game was represented in the ASP input. Additionally, the implementation was much slower than desired. Because ASP based systems have been found to handle large-scale generation problems well, this issue was likely also a result of the intricacies surrounding how the rules were denoted for the ASP [Lin18]. Such results highlight the importance of defining clear game logic in the start to avoid such undesirable features for our level generator.

2.3 Wave Function Collapse Based PCG

The Wave Function Collapse algorithm was originally an example-driven image generation algorithm. It was later expanded into the world of PCG and used for level generation. It uses a non-backtracking, greedy approach that looks to match each “window” present in the input directly to a “window” present in the output [KS17]. The algorithm can be understood in a constraint solving light. In this way, it is essentially weighing certain heuristics and selecting the best or minimum values. It relies heavily on the heuristic of the minimum remaining value. The algorithm has been found to work best with abstract chunks of input rather than more literal input. Various versions of the algorithm have been formulated since its release including work to add in backtracking capabilities and building out a complete Unity tool set asset for it [KS17]. Figure 4 shows an example of how the Wave Function Collapse
algorithm uses pattern matching for content generation. Here we can see the parsing of tiles from the input image. It takes these tiles and the rules for what they can be adjacent to and produces a new image for which the rules are satisfied.

Figure 4: Example of Wave Function Collapse pattern matching.

Kim et al. introduced a use of the Wave Function Collapse algorithm that can work with graph based representations of content rather than simply grid based content [KLL+19]. This research successfully altered the algorithm to accurately produce content as specified. To test this graph based version of the Wave Function Collapse algorithm they applied it to generate game content for various game styles including Sudoku and 3D prototype game levels. In the generation of 3D prototype game levels, they made use of an open source path-finding algorithm and the 3D software Blender.

Success with this style of level generation indicated that implementing the graph based version of the algorithm may offer a great reference for our level generator as it will need to produce 3D based Monument Valley style game levels. The results of the experiments with both Sudoku and the 3D prototype levels indicated that this new version of the Wave Function Collapse algorithm allows for PCG to take advantage of Wave Function Collapse’s excellent content control [KLL+19]. However, a potential drawback for this implementation was the growth in computation time once it was paired with graph based representation. To improve such computation time, work with optimization must be done since the graph based representation requires many more constraints to be specified at the start of the algorithm [KLL+19]. Additionally, the Wave Function Collapse algorithm required a sufficiently large example from which to extract the tiles or “windows” on which to frame its matching based generation. Our inventory of Monument Valley styled game levels may not satisfy this requirement.

2.4 Evolutionary Search Based PCG

Evolutionary search based methods for PCG take inspiration loosely from the concept of biological evolution and the concept of survival of the fittest [TS16]. Generally, it starts with a random “population” of levels. Each level is judged by some evaluation function and ranked by their resulting values. The top-ranked levels remain in the “population” while the rest are substituted for mutated or reproduced versions of the highly ranked levels. This process continues until some level in the “population” reaches a predetermined goal score from the evaluation function [TS16].
The key features of this method are the search algorithm, the content representation, and the evaluation function [TS16]. Tweaking each of these features can improve or harm the efficiency and success of this method. Typically the game space is initially represented abstractly in a tree or graph and is evolved to a more concrete representation over the course of each iteration [TS16]. In addition to impacting the performance of the algorithm (i.e. the search speed, game object creation ability, etc.), the style of representation can also impact the final appearance of the level [ART16].

Evolutionary search based content generation has been used to produce levels within a variety of puzzle games. Specifically, Kartal et al. applied it to procedurally generate Sokoban styled puzzle game levels [KSG16]. Sokoban is a puzzle game that requires the player to move boxes within the game environment to certain goal positions. See Figure 5 for an example of the game interface.

The green arrows indicate the goal path for the generated level. The paper’s particular implementation of this form of PCG depended on the Monte Carlo Tree Search algorithm. The authors structured their game environment as a tree and defined a set of actions by which to evolve the tree to a playable game level. When a terminal action from the rule set was performed the game level was evaluated by a data-driven evaluation function. The authors developed this evaluation function based on the data gathered from human input. This human input ranked the difficulty of existing Sokoban levels. Comparing their generated levels against this input the authors could determine if their levels had reached the desired level of difficulty [KSG16].

Results from generating levels in this way did successfully generate playable levels with increasing difficulty. However, the authors noted that generating larger puzzle levels caused the time required for the level generation to grow exponentially. From these results, we learn that using tree-based content representation and data-driven evaluation functions are successful aspects for search based PCG methods when puzzle levels have a relatively small game environment. Our Monument Valley styled levels range in size and complexity. This approach will likely demonstrate similar difficulties when attempting to produce such levels.

A similar search based method was performed by Baghdadi et al. with a Spelunky styled game [BEAO+15]. Spelunky, while not directly a puzzle game, requires puzzle-like path planning to “solve” its levels. Their implementation of search based PCG relied on
a Genetic Algorithm and node based graphs to perform level generation. Figure 6 shows how this method evolved its generated levels from an abstract representation to actually indicate what the game level should include, as noted by the legend to the right. Unlike the Sokoban based level generation, this approach did not use a data-driven evaluation function. The authors used a function based on comparing a starting difficulty score to all the evolved difficulty scores. The results indicated that this approach could also successfully create playable levels, yet the paper identified that there was little variation between the procedurally generated levels. The paper cited the potential of distributing “enemies and items” within the generated game space post-generation to “spice up the level” [BEAO+15]. This idea of adding post-generation “spice” offers potential in how we can use our own Monument Valley style “enemies” and game features (e.g. crows, building windows, and rooftop flags as seen in Figure 1) to ensure that our generated levels do not reflect the same lack of variation that this implementation identified.

2.5 Test-and-Reject Based PCG

Test-and-reject based PCG is an approach to content generation that does as it says. Essentially, it relies on an “expert” or designer provided template that the system, then, uses to generate levels [DB17]. Each generated level is tested by a constraint checker to ensure it meets the requirements of a sound game level. If the levels fail any number of the tests in this check, the level is rejected and the generation step begins again. This process loops until a level is found to pass all the tests from the constraint checker. This approach is a slightly more rudimentary version of the evolutionary search based method. Both approaches require a step for testing or evaluating the generated content. The primary difference is that with test-and-reject the levels that do not satisfy the test are simply rejected while with evolutionary search based PCG the levels that do not rank highly by the evaluation function are evolved into levels reflecting the properties of the levels which the evaluation function ranks higher [TS16].

Dong and Barnes implemented a test-and-reject based method for procedurally generating educational puzzle game levels for the game BOTS, see Figure 7 [DB17]. The red circled sections indicate the key aspects of the BOTS game interface including the game environment and the player movement options. In this study, the authors looked to test-and-reject PCG to allow for puzzle levels to be generated faster than relying on human “experts” to produce puzzles on an ad hoc basis. Specifically, this approach was careful to ensure the PCG system still allowed for the generated levels to maintain the game’s particular educational intent. With this in mind, the authors selected a test-and-reject based PCG method such that the experts who typically
produced the game levels could instead simply input a template with encoded goal requirements. The system first parsed the provided template, checking the template’s validity, and then created a template object. This object was passed to the program generator which converted the object into a valid solution program. This program was tested by the constraint checker to ensure that the requirements for a “good” game level had been met. If not, the program was rejected and the program generator had to run again to produce a new solution program. This process was repeated until a solution program was generated that passed the tests of the constraint checker. Finally, the puzzle file formatter converted the program into a puzzle file that was compatible with the BOTS game.

The results of this method successfully produced valid puzzle levels for the encoded goal requirements. Dong and Barnes additionally identified that the generated levels were produced much faster than the expert produced levels were for the same goals. Further, the generated levels had a higher level of variation. However, the generated levels had less pattern apparentness than the expert levels. In the realm of educational games, where specific learning goals are the primary focus, this could lead to the intended learning goal not coming across clearly. Similarly, this test-and-reject based PCG system did not allow for difficulty constraints to be considered. For *Monument Valley* styled puzzle levels such issues would be important to remedy. Despite not being an educational game, *Monument Valley* does have clear goals for the player to “learn” from each level and these would ideally be fairly apparent during gameplay. In addition, the difficulty of the generated levels for *Monument Valley* styled levels is important for realistic and interesting progression between each level.

2.6 Influences on Our Implementation

Each of these PCG methods offer pros and cons. The specific use of each method in the work and papers outlined offer additional insight into how one might best approach level generation for puzzle games similar to *Monument Valley*. Assessing these conclusions, as well as our inventory size and timeline, we propose a method for generation most similar to that of Test-and-Reject. Like Dong and Barnes’ BOTS levels [DB17], *Monument Valley* styled levels have clear goals to achieve during gameplay and direct rules for how the game environment can be structured. We used these goals and rules as a template for which the system based its level construction. Each produced level was directly constrained by how the game environment could feasibly be built so this approach focused more on testing and required little rejection consideration.

3 Methodology/Efforts To Date

The focus of this project was determining how to develop an abstract representation of the *Monument Valley* styled puzzle levels and, then, how to best generate these abstractly represented levels. The content representation must account for the architecture present in the game environment, the “switches” used to alter the game environment, as well as, the level’s goal/intended path points. The preliminary work in this endeavor was recreating several levels that currently exist in the actual *Monument Valley* commercial game. Such
work required Unity to build out the game architecture and C# to program the “switches” present within the level. The process for recreating these aspects for each level followed a similar workflow outlined below.

First, a level was chosen from the game based on the architecture present and the “switching” style used. Because we required a variety of “switches” to generate levels with variation, we had to ensure that the hand built levels exhibited this variation as well. Once an existing reference level had been selected, the architecture within the level was modeled in Unity. The root shape within the game architecture was a cube. Long thin cylinders were also used to create supports within double-decked levels. As a placeholder, the game character was represented by a red sphere. Building out levels required determining the placement of a series of cubes such that they represented the desired shape within the isometric view that Monument Valley makes use of. See Figure 8 for a side-by-side comparison of the scene view in Unity and the isometric game view. The scene view depicted the game environment as the level designer saw it while the isometric game view depicted the level as the player would. Ensuring that the two views created the desired shape/environment posed some challenges as the isometric view can make objects appear to be in a physical space they were not actually in from the designer’s perspective in the scene view. Mapping the placement from the existing levels into the recreated levels required an understanding of how the placement will lend itself to the level’s use of Escher-like “switching.” Additional consideration was given to the object materials in the built game environment. A shader was created to include a distinguishing outline along each edge and apply color evenly across each cube and cylinder. For particular objects, materials to denote the presence of ladders, doors, and goal positions were also included.

Figure 8: Example of a constructed level.
When the architecture of the built game environment matched that of the existing level, the work shifted focus to programming the “switches” necessary for the level. The “switches” were encoded in C# and essentially animated the game objects to move in such a way that new path potentials were revealed and/or advanced the game level. These “switch” animations made use of simultaneous rotations and translations. In the current state, these “switch” animations are cued by user key presses.

The game level could have multiple “switches” required to complete the intended path. For the level in Figure 9, there were a total of six “switches” involved. The series of “switches” and their impact on the configuration of the game environment is shown in Figure 9. Using the “switches” in the order from Figure 9a to Figure 9g allowed the game character to reach the upper target position. The first “switch” (Figure 9b) rotated the purple highlighted section to form a ramp. The second “switch” (Figure 9c) rotated the central object to meet the left upper platform. The third “switch” (Figure 9d) rotated the central object to meet the far right platform on which a button appears. The fourth “switch” (Figure 9e) pressed the button on the far right platform down, which triggered the purple highlighted stack of three cubes to fall to the left. The fifth “switch” (Figure 9f) rotated the purple highlighted central object to meet this back platform. This “switch” demonstrated how the Escher-like
illusions are used in gameplay. Now the game character could reach the upper deck of the purple highlighted central object from the lower deck. The final “switch” rotated the purple highlighted central object one last time such that its upper deck met the target goal position. In combination, these “switches” allowed the player to move their game character, the red sphere, from the start position it is seen in now to the goal position marked by the square target. Without activating these “switches,” occupying this goal position would have been impossible for the user.

After all the “switches” for the existing level were present in the hand built level, all the necessary pieces were represented and this recreation process could begin for the next level. This workflow was repeated until all the desired architecture components and “switches” had been recreated in the hand built levels. We approximated this requiring five hand built levels, each with unique game components.

With these constructed game levels, an inventory of the “switches” and game objects could be compiled. From the five hand-built levels, twelve total “switch” styles were identified: ladders, doors, simple translations, linked translations, simple rotations, linked rotations, open-faced rotations, rotation + camera direction movement, rotation + translation, 45° cube divisions, buttons, and button reveals. Each of these “switch” styles included their own requirements and could only be placed on specifically shaped game objects.

Additionally, from the five hand-built levels, a variety of game objects were compiled. These game objects were groups of cubes and cylinders parsed from the larger built game environments that could be packaged separately and included within other levels. Each of these game objects had identified “switch” styles for which they were compatible.

These two inventories formed the root for our level generator. From here we specified rules for how and where the compiled game objects could connect. These rules acted similarly to the template within the test-and-reject methods described above. We could then allow for the rules to guide a series of random selections of game objects and corresponding “switches.” Once the guided random selecting concluded, a new level that adhered to the rules was produced.

Current efforts are focused on polishing the hand-built levels, as well as parsing and compiling these two inventories. Once these processes are complete, the focus will shift to defining the rule for how and where the game objects can connect. From here we can begin the actual generation of new game levels. Entering this phase will mean a large amount of testing. We will inspect the guided random selections and adjust or add rules to our “template” as needed. In doing this testing we will look for results that are obtained efficiently, are solvable, demonstrate enough variation, and make interesting use of the Monument Valley style “switches”.

References


