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 look to a specific path-puzzle game, called Monument Valley, for a basis to begin experimentation. Specifically, the work for the project requires 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 Evolutionary Search Based Methods for PCG .................. 8
   2.2 Grammar Based PCG ............................................. 9
   2.3 Test-and-Reject Based PCG ...................................... 10
   2.4 Answer Set Programming Based PCG .......................... 11
   2.5 Wave Function Collapse Algorithm Based PCG .............. 12

3 Methodology/Efforts To Date ....................................... 13
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, much 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 evolutionary search [TS16], grammars [TSD16], test-and-reject [DB17], answer set programming [NS16], and wave-function collapse [KS17]. 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 [SLT+16] 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 commercial levels look like. 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. Focusing on the Monument Valley style of puzzle games involves a new set of problems for the level generation beyond those normally involved in the 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. 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

Figure 1: Monument Valley Reference Image.
style of PCG algorithm will produce playable, fun levels of this kind efficiently. The result of this project is such an algorithm. With it, one can produce playable puzzle game levels of the *Monument Valley* style. 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 originate 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 can see this in the following analyses of some of the major types of PCG algorithms for level generation.

### 2.1 Evolutionary Search Based Methods for 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. apply 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. Their particular implementation of this form of PCG depends on the Monte Carlo Tree Search algorithm. They structure their game environment as a tree and define a set of actions by which to evolve the tree to a playable game level. When a terminal action from the rule set is performed the game level is evaluated by a data-driven evaluation function. They develop this evaluation function based on the data gathered from human input. This human input ranks the difficulty of existing Sokoban levels. Comparing their generated levels against this
input they can determine if their levels have reached the desired level of difficulty [KSG16]. Their results from generating levels in this way did successfully generate playable levels with increasing difficulty. However, they note that generating larger puzzle levels causes 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 relies on a Genetic Algorithm and node based graphs to perform their level generation. Unlike the Sokoban based level generation, this approach did not use a data-driven evaluation function. They use a function based on comparing a starting difficulty score to all the evolved difficulty scores. Their results indicate that their approach can also successfully create playable levels, yet they identified that there was little variation between their procedurally generated levels. They 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 identifies.

2.2 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 means the inclusion of ‘[’ and ‘]’ in the symbols used. ‘[’ saves the current position within the string and ‘]’ returns to that previously saved position in a “push” and “pop” way [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 complete generated level [TSD16].

Grammars have a wide range of applications within PCG. Jemmali et al. use it to proce-
durably generate levels for their education focused puzzle game, May’s Journey [JICEN20]. They let the grammar based method handle the content generation aspect of their level generation while implementing a “work backward” approach to ensure the generated levels are solvable as desired without needing post-processing or filtering. Specifically, they convert their input, a random solution, to an abstract syntax tree. From the tree, they easily extract the game object that should be involved in the level, as well as the actions and attributes ascribed to it. This is passed through the grammar that uses a series of rules to determine what shape the game environment should take to satisfy all the requirements. The rules within the set are weighted with probabilities to favor certain rules over others. Additionally, penalties are implemented when rules are selected so that no one rule is disproportionally chosen. Each object specific map shape is combined to a final game space by minimizing the number of modifications needed for such merging. Finally, either a reward path algorithm or a maze path algorithm is used to determine the path through the level. Either algorithm maintains entrance-to-exit navigability [JICEN20].

The results of the grammar based PCG approach efficiently produced solvable levels with a high amount of variation. They identify, however, that their method requires 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 presents the opportunity to make use of the highly successful model while avoiding this identified dependence on coding conventions as the gameplay within Monument Valley styled games consists of fairly simple point-and-click environment navigation.

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.

2.3 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 did rank highly by the evaluation function [TS16].

Dong and Barnes implement a test-and-reject based method for procedurally generating educational puzzle game levels for the game BOTS [DB17]. In this study, they look to such PCG to allow for puzzle levels to be generated faster than relying on human “experts” to produce puzzles on an ad hoc basis. Specifically, they were careful to ensure their PCG system still allowed for the generated levels to maintain their particular educational intent.
With this in mind, they selected a test-and-reject based PCG method such that the experts who typically produce the game levels can instead simply input a template with encoded goal requirements. The system first parses the provided template, checking the template’s validity, and then creates a template object. This object is passed to the program generator which converts the object into a valid solution program. This program is tested by the constraint checker to ensure that the requirements for a “good” game level have been met. If not, the program is rejected and the program generator must run again to produce a new solution program. This process is repeated until a solution program is generated that passes the tests of the constraint checker. Finally, the puzzle file formatter converts the program into a puzzle file compatible with the BOTS game.

The results of this method successfully produced valid puzzle levels for the encoded goal requirements. Dong and Barnes additionally identify 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.

2.4 Answer Set Programming Based PCG

Answer set programming (ASP) is a logic programming approach for PCG [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 is outlined by Lindeman to generate levels for a self-created puzzle game called Swappy [Lin18]. In this application of ASP, the method of “core-sim-style” [Lin18] is implemented in which three AnsProlog programs are developed. One, core, is used to generate the game pieces and explain the relationship between them. The second, sim, acts as a simulation of gameplay to ensure the rules of gameplay are followed. The final, style, is used to generate artifacts with a mind towards some aesthetic criteria. These three programs are linked by a Python command-line tool that takes in parameters defining the level width and the number of character tokens to be used in the level. The AnsProlog files are run by the Clingo ASP solver. Of the many output levels, one is randomly selected once each ASP program has run. The Python tool then parses and renders the facts of the answer
set chosen into ASCII art in the terminal. This output is finally provided to the Swappy game client [Lin18].

The results of this implementation of ASP were mixed. Lindeman notes that the generator occasionally still produces unsolvable levels. This is 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 is 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.

2.5 Wave Function Collapse Algorithm 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].

Kim et al. introduce the 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 make use of an open source path-finding algorithm and the 3D software Blender.

Success with this style of level generation indicates 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 indicate 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 to consider for this implementation is the way that the calculating time grew once it was paired with graph based representation. To improve such calculation 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 requires a sufficiently large example from which to extract the tiles or “windows” on which to base its matching based generation. This requirement may not be satisfied by our inventory of Monument Valley styled game levels.
3 Methodology/Efforts To Date

The focus of this project is determining how to develop an abstract representation of the *Monument Valley* styled puzzle levels and, then, how to best generate these abstractly represented levels. How the content is represented 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 is recreating several levels that currently exist in the actual *Monument Valley* commercial game. Such work requires 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 follows a similar workflow outlined below.

![Scene view in Unity.](image1.png) ![Isometric game view in Unity.](image2.png)

Figure 2: Example of a constructed level.

First, a level is chosen from the game based on the architecture present and the “switching” style used. Because we require a variety of “switches” to generate levels with variation, we ensure that the hand built levels exhibit such variation as well. Once an existing reference level has been selected, the architecture within the level must be modeled in Unity. The root shape within the game architecture is a cube. Long thin cylinders are also used to create supports within double-decked levels. As a placeholder, the game character is represented by a red sphere. Building out levels requires determining the placement of a series of cubes such that they represent the desired shape within the isometric view that *Monument Valley* makes use of. See Figure 2 for a side-by-side comparison of the scene view in Unity and the isometric game view. The scene view depicts the game environment as the level designer sees it while the isometric game view depicts the level as the player would. Ensuring the two views create the desired shape/environment poses some challenges as the isometric view can make objects appear to be in a physical space they are not actually in from the designer’s perspective in the scene view. Mapping such placement from the existing levels into the recreated levels requires an understanding of how the placement will lend itself to the level’s
use of Escher-like “switching.”

When the architecture of the built game environment matches that of the existing level, the work shifts focus to programming the “switches” necessary for the level. The “switches” are encoded in C# and essentially animate the game objects to move in such a way that new path potentials are revealed and/or advance the game level. These “switch” animations make use of simultaneous rotations and translations. In the current state, these “switch” animations are cued by user key presses.

The game level can have multiple “switches” required to complete the intended path. For the level in Figure 2, there are a total of four “switches” involved. To date, three of these have been programmed into the built level. The series of “switches” and their impact on the configuration of the game environment is shown in Figure 3. The first “switch” (Figure 3b) rotates the central elbow. The second “switch” (Figure 3c) raises the central elbow to the second deck of the environment. The third “switch” (Figure 3d) rotates the central elbow in such a way that the first deck of the environment appears to now lead to the open space between the decks. This “switch” demonstrates how the Escher-like illusions are used in gameplay. The final “switch” will rotate the central elbow one last time in such a way that the first deck appears to be able to access the second deck of the environment. Without activating these “switches,” occupying this goal position would be impossible for the user.

After all the “switches” for the existing level are present in the hand built level, all the necessary pieces are represented and this recreation process can begin for the next level. This workflow is repeated until all the desired architecture components and “switches” have been recreated in the hand built levels. We approximate this requiring ten hand built levels, each with unique game components. With these constructed game levels, an inventory of the “switches”, game objects, and paths can be compiled. These inventories will be the input for our level generator.

Current efforts are focused on this process of hand done level recreation in Unity and C#. Once this building and encoding are complete, the focus will shift to testing the styles.
of PCG level generation described above. This process will involve determining how to best represent the items within our inventories. As seen in the related works, the representation chosen will likely be determined by the method of generation chosen. 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


