Level Generator for *Laserverse* using ASP

Abhijeet Krishnan
North Carolina State University
akrish13@ncsu.edu

Abstract

Procedural level generation is a great way to create more content and unexpected scenarios within a game (Davis 2017). We demonstrate a level generator built for the puzzle game *Laserverse* using Answer Set Programming (ASP). The design of the generator is explained and design decisions are justified. An evaluation of the generator is presented using playability as a metric. We discuss the challenges faced while writing the generator as well as possible avenues for improvement - both of the generator as well as the process of writing a generator using ASP.

Introduction

Procedurally generated levels are a major aspect of PCG research since they have the potential to allow almost any game to be infinitely replayable. However, level design is a complex and creative discipline in which current procedural level generators generally fail to match up to human level designers. Togelius et. al. (Togelius et al. 2013) describes some of the problems seen in generated levels for *Super Mario Bros.*, namely a lack of progression and macro-structure, a lack of storytelling or teaching, inexplicable structures and content, difficulty spikes and repeated structures. While we recognized that the field is ripe for greater advancement, we settled on the modest goal of simply designing a working generator for our game.

Laserverse is a puzzle game developed in PuzzleScript by Martens et. al. (Martens et al. 2018) for the purpose of studying the theory of mental model matching. The hypothesis of this work was that players learn puzzle game mechanics through an iterative process of hypothesizing, failure and revision of mental models. The game was designed with a number of mechanics designed to interact in a wide array of combinations. Our goal was to support this work by designing a level generator which offered control over the specific mechanics used in the levels, enabling the authors to test their hypothesis on a wider variety of participants without having to hand-author multiple levels.

We also present an evaluation of our generator using the metrics of *playability* and *solution length*.

Copyright © 2018, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Background

Level Generation using ASP

ASP is a programming paradigm (like procedural or functional programming) geared towards solving NP-hard search problems. The problem domain is specified as a set of rules (or facts) and stable models are generated which satisfy all those rules. If we can specify our game's level design space as a set of ASP constraints, we can use a solver to generate solutions which satisfy those constraints, each of which is a playable level. We use clingo, part of the Potassco answer set solving collection (Gebser et al. 2011) as our solver of choice.

Smith and Mateas' paper (Smith and Mateas 2011) was the first one to argue in favour of ASP as a useful tool for procedural content generation, citing ASP's brevity, expressiveness and generality in quickly constructing design spaces for games. Later work by the authors on a puzzle game Refraction (Smith et al. 2012) was also useful, since the game shares some similarities with *Laserverse*. A book chapter by the same authors (Nelson and Smith 2016) provides more instruction in how to write generators for mazes.

Laserverse

Laserverse is a puzzle game built using the PuzzleScript game engine (Lavelle 2013). It offers a simple rule-based syntax for describing actions in the game world. For example, say we want the player to push a crate if the player walks into it. We can express this succinctly in PuzzleScript as

```
[ > Player | Crate ] -> [ > Player | > Crate ]
```

Laserverse is centered around the idea of interacting with lasers and mirrors in game to power sensors, which then open doors to the exit. There are additional mechanics such as buttons, crates, splitters and logic gates. A full explanation of each mechanic in the game can be found in the original paper, with mechanics missing from the original paper described in Figure 2. The mechanics for *Laserverse* are expressed in about 121 lines of PuzzleScript.

Generator Evaluation

The idea for evaluating the design space of a generator was first proposed in Smith and Whitehead's (Smith and Whitehead 2010) seminal paper on the topic, in which they also presented an evaluation of a level generator for their game

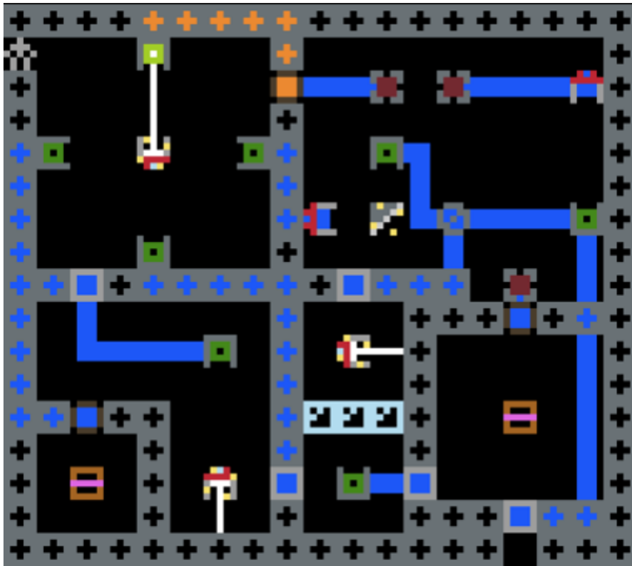

Figure 1: Screenshot of *Laserverse* level *Wire-fu* (hand-authored)

Launchpad. The idea is to define comparison metrics using which generated levels can be compared. The metrics chosen should reflect global properties of levels and aid in understanding a generator's expressive range. With these metrics, we can then visualize how the expressive range of a generator changes when modifications are made to the generator.

For our generator, we define the comparison metrics of *playability* and *solution length*. Playability is a boolean value which is True if the level is playable i.e. a solution exists, and False if it is not. Solution length is a measure of the minimum number of steps required to solve a certain level.

Approach

Explain what you did in sufficient detail that someone could reproduce your work. High-level algorithm descriptions, equations, and descriptions of your implementation decisions are all in-scope. Explain how you evaluated your work.

Defining a design space

The highly useful approach to defining the design space of all possible levels would be to restrict ourselves to playable levels. This would require a notion of *playability*. As will be described later, coming up with this notion for a game like *Laserverse*, with so many interacting mechanics, proved too difficult.

To simplify the design space, we opted to restrict ourselves to levels with a fixed kind of solution. We identified three such solution types.

1. Player picks up crate - player drops crate on button - button opens door to exit - player walks to exit
2. Player rotates laser - laser activates sensor - sensor activates wires to door - door opens to exit - player walks to exit
3. Player rotates laser - laser reflects off mirror - reflected beam activates sensor - sensor activates wires to door - door opens to exit

This leads to a lot of simplifications. We only need one of each object for whichever level we generate (e.g. one button, one crate and one door for level of type 1). We can define playability in terms of the existence of paths between each objects in the solution. We can ignore all other mechanics and object types present in *Laserverse*.

We follow the generate and test methodology to develop our generator. We generate all possible levels with the line -

```
{ at(X, Y, T) :tile(T) } = 1 :- dimX(X), dimY(Y).
```

This tries every possible tile (object) type for every possible tile location in the game. This is clearly just a brute force enumeration of every possible level in the design space, so we refine this design space with additional rules.

We define cardinality constraints for each type of object in the game. This allows a designer to change the number of these objects in generated levels. Given our design space, we would only get playable levels if we fix the number of objects as 1. The code for this can be seen below.

```
#const minLasers = 1.
#const maxLasers = 1.
minLasers { at(0..n-1, 0..m-1, T) :laser(T) }
    maxLasers.
```

We also constrain our levels to a 6×6 grid to reduce generation time. This can be modified, but generating a level would take longer.

We add some rules to ensure our design space matches the authored levels. For example, we add rules to ensure that the border is a wall, that the player and exit both lie on the border. To enforce playability, we require that the player and exit not lie on a corner.

We then add rules specific to each level type. For example, in type 2, we add rules which prevent the laser from being blocked by anything on its path to the sensor.

Now by enabling the rules for each level type in turn, we can generate levels of each type. These levels are not guaranteed to be playable, since the paths to various objects might be blocked, and we have not added rules which explicitly disallow this (due to the difficulty encountered when writing such rules).

Evaluation

The choice of suitable comparison metrics for puzzle games is an open question. One possible metric is the sense of surprise experienced by players when they discover how mechanics interact in unexpected ways to lead to a solution, but it is tricky to quantify this. In any case, we have very few mechanics we choose to work with for our evaluation, since we want to have a large number of playable levels.

Therefore, we opt for the relatively simple metrics of playability and solution length to differentiate our levels. We measure solution length only for playable levels. We have written a DFS-based agent in Python to solve our generated

Movable Laser	Like the basic laser, but the player can move it around the game world without changing its orientation
Movable Mirror	Like the basic mirror, but the player can move around it the game world without changing its orientation
Crate	Can be moved around the game world and dropped onto buttons to keep them activated
Splitter	Splits an incoming beam of light into two perpendicular beams
Rotatable Splitter	A splitter that the player can rotate to choose where the split beams are incident.
Movable Splitter	Like the basic splitter, but the player can move it around the game world without changing its orientation
AND Gate	Activates only if both inputs are activated
Open Door	A door which is open by default, and closes when activated
Glass Pane	A transparent block which is solid but allows laser beams to pass through it
Sensor	Activates a wire if a laser beam is incident on it
Guard	Prevents a laser beam from striking a goal, usually allows beams from a single direction only

Figure 2: Elemental mechanics in *Laserverse*

levels to measure these metrics. The complete code for our generators and evaluators can be on found on GitHub[1].

Results

Playability

	Playable	Unplayable	Total
Type 1	548	452	1000
Type 2	466	534	1000
Type 3	537	463	1000

Figure 3: Number of playable levels for each level type

Solution Length

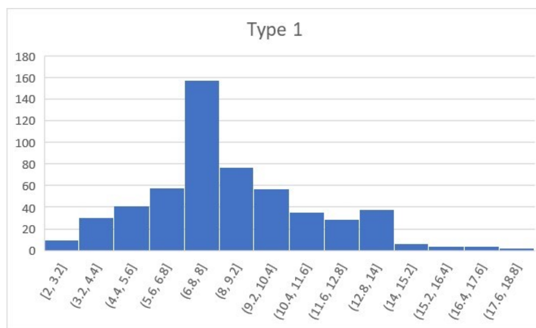

Figure 4: Histogram of solution lengths for type 1 level

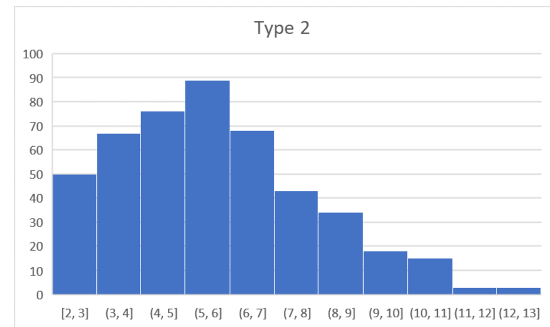

Figure 5: Histogram of solution lengths for type 2 level

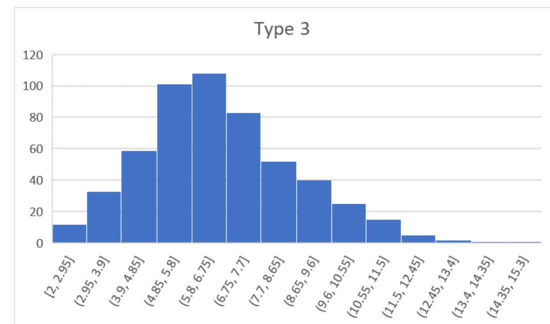

Figure 6: Histogram of solution lengths for type 3 level

Discussion

Challenges

Lack of expressivity A general notion of playability is hard to accurately capture for *Laserverse*. We can look to another puzzle game called Sokoban, which has also been implemented in PuzzleScript. A generator for Sokoban was implemented by the team behind clingo for the 3rd Answer Set Programming Competition (Calimeri et al. 2011) using time as a variable to represent a plan of actions. However, Sokoban has only a few actions, while *Laserverse* has many, depending on the objects in the environment. Therefore, a notion of playability would involve finding a plan of actions which would solve a given level. This is difficult to code using ASP. Indeed, the simple notion of having a single wire connect two objects was also difficult to conceptualize and code. This could be due to the author's inexperience with ASP.

Lack of software engineering tools Common tools such as a debugger have no equivalent while working with ASP. Say a certain new rule is causing undesired answer sets to be produced, or more commonly, causing the solver to return an UNSATISFIABLE verdict. It is difficult to pinpoint why ex-

[1] `https://github.ncsu.edu/akrish13/laserverse-level-generator`

actly this happens without a deep knowledge of ASP or logic programming. Tools which expose the inner working of the solver better would be useful here. There has been some work on software engineering for ASP (Febbraro, Reale, and Ricca 2011), but there was no tool which proved useful to us.

Additionally, since the codebase for this generator was uncommonly large, it would be helpful to have a facility similar to make for C/C++ programming which can utilize rules written in different files together. This would aid modularity and ease while developing generators using ASP.

Levels generated using ASP also suffer from lack of intent and storytelling.

Improvements and Future Work

While additional and more complex level types can always be constructed, the ultimate goal would be to capture a general notion of playability in ASP. The idea of using mission generation to generate a desired solution using ASP, and then using ASP to embed that in a puzzle grid layout seems promising.

References

Calimeri, F.; Ianni, G.; Ricca, F.; Alviano, M.; Bria, A.; Catalano, G.; Cozza, S.; Faber, W.; Febbraro, O.; Leone, N.; Manna, M.; Martello, A.; Panetta, C.; Perri, S.; Reale, K.; Santoro, M. C.; Sirianni, M.; Terracina, G.; and Veltri, P. 2011. The third answer set programming competition: Preliminary report of the system competition track. In Delgrande, J. P., and Faber, W., eds., *Logic Programming and Nonmonotonic Reasoning*, 388–403. Berlin, Heidelberg: Springer Berlin Heidelberg.

Davis, G. 2017. Procedural level generation in unity for M.E.R.C. (part 1 of 2). `https://www.gamasutra.com/blogs/GrahamDavis/20170130/290326/Procedural_Level_Generation_in_Unity_for_MERC_part_1_of_2.php`. [Online; accessed 07-December-2018].

Febbraro, O.; Reale, K.; and Ricca, F. 2011. Aspide: Integrated development environment for answer set programming. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, 317–330. Springer.

Gebser, M.; Kaufmann, B.; Kaminski, R.; Ostrowski, M.; Schaub, T.; and Schneider, M. 2011. Potassco: The potsdam answer set solving collection. *Ai Communications* 24(2):107–124.

Lavelle, S. 2013. Puzzlescript. `https://www.puzzlescript.net/index.html`. [Online; accessed 07-December-2018].

Martens, C.; Williams, A.; Alexander, R. S.; and Dabral, C. 2018. Generating puzzle progressions to study mental model matching. EXAG 2018.

Nelson, M. J., and Smith, A. M. 2016. Asp with applications to mazes and levels. In *Procedural Content Generation in Games*. Springer. 143–157.

Smith, A. M., and Mateas, M. 2011. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games* 3(3):187–200.

Smith, G., and Whitehead, J. 2010. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, 4:1–4:7. New York, NY, USA: ACM.

Smith, A. M.; Andersen, E.; Mateas, M.; and Popović, Z. 2012. A case study of expressively constrainable level design automation tools for a puzzle game. In *Proceedings of the International Conference on the Foundations of Digital Games*, 156–163. ACM.

Togelius, J.; Champandard, A. J.; Lanzi, P. L.; Mateas, M.; Paiva, A.; Preuss, M.; and Stanley, K. O. 2013. Procedural content generation: Goals, challenges and actionable steps. In *Dagstuhl Follow-Ups*, volume 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.