Grand Valley State University
ScholarWorks@GVSU

Technical Library School of Computing and Information Systems

2018

Puzzle Level Generation with Answer Set Programming

Daniel Lindeman
Grand Valley State University

Follow this and additional works at: https://scholarworks.gvsu.edu/cistechlib

ScholarWorks Citation
Lindeman, Daniel, "Puzzle Level Generation with Answer Set Programming" (2018). *Technical Library*. 325.
https://scholarworks.gvsu.edu/cistechlib/325

This Project is brought to you for free and open access by the School of Computing and Information Systems at ScholarWorks@GVSU. It has been accepted for inclusion in Technical Library by an authorized administrator of ScholarWorks@GVSU. For more information, please contact scholarworks@gvsu.edu.

Puzzle Level Generation with Answer Set Programming

By
Daniel Lindeman
December, 2018

Puzzle Level Generation with Answer Set Programming

By
Daniel Lindeman

A project submitted in partial fulfillment of the requirements for the degree of
Master of Science in
Computer Information Systems

at
Grand Valley State University

December, 2018

____
Dr. Robert Adams **Date**

Table of Contents

Abstract 3

Introduction 4

Background and Related Work 5

Program Requirements 5

Implementation 6

Results, Evaluation, and Reflection 8

Conclusions and Future Work 9

Bibliography 10

Appendices 11

- **Appendix A** 12
- **Appendix B** 13
- **Appendix C** 14

Abstract

Swappy is a puzzle game that requires different character tokens to cooperatively navigate a maze to reach their goals. Swappy characters are special in that whenever they are collinear with another character, they may swap places. In practice, generating levels manually may take upwards of 20 hours and is error prone. After exploring the space of procedural content generation, it was hypothesized that by employing Answer Set Programming (ASP), it would be possible to generate and constrain level creation such that procedurally generated levels are solvable, meet an aesthetic standard, and follow the rules of the game. Using the grounder/solver tool, Clingo, level creation can be done in a matter of seconds or minutes. The expressive power of rules and constraints allows the developer to more clearly see their game for the abstract ruleset that it is. In this project, we explore the use of AnsProlog to generate artifacts useful for level generation for the puzzle game Swappy - finding succinct and expressive ways to do so compared to traditional programming languages.

Introduction

Swappy is an puzzle game I developed several years ago. Swappy is a maze game at its core, with some special caveats regarding character traversal and cooperation. In Swappy puzzles, there are two to four character tokens of different colors. To solve a level, all character tokens must reach their like-colored goal at the same time. Character tokens may move to adjacent spaces, and may walk through doors that share their color. The eponymous mechanic comes into play when character tokens are collinear on the map. For example, if the green and blue character's X coordinates are the same, the two tokens can switch places with each other. The same swapping rules are true for matching Y coordinates.

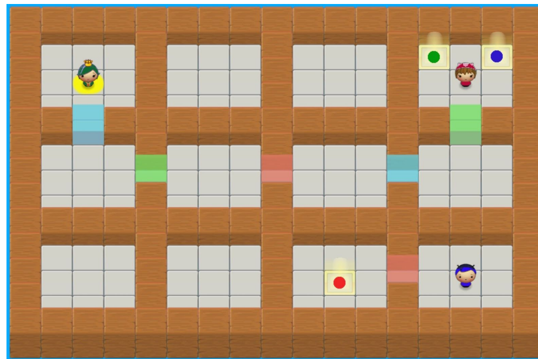

Figure 1. A Swappy level with three character tokens.

In its early stages, Swappy levels were generated using buttons, graph paper, and Sharpie markers. The process for making a single level would take several hours. The experience of manually generating levels this way was enough to convince me to work on a procedural content generation (PCG) system for Swappy. Often, after working on a level, playtesting would reveal shortcuts and other undesirable errors. Because I also author the levels to be compact, it is difficult to correct errant pathways without removing the intended solution or introducing even more errors. This project set out to discover and develop methods for procedurally generating content for the game, in order to: increase the speed of authorship; reduce errors; and lower frustration.

Background and Related Work

AnsProlog is an expressive logic programming language in the realm of content generation. Using facts and rules, a developer describes their game, and only their game. AnsProlog will then do the heavy lifting and create a program consistent with those facts and rules. Compared to coding in a traditional language, like Python or Java, where the developer is left to explore minutiae in lieu of addressing their core problem. For example, when I began this project with an agent-based approach (detailed later), the approach involved code for mutating matrices, checking boundary conditions, and text parsing - none of which have much to do with the rules of Swappy.

A subset of logic programming, Answer Set Programming (ASP), uses facts, rules, and constraints to create stable models given input programs. AnsProlog is a dialect of Prolog used for ASP. Unlike standard Prolog, which generates a single consistent space from the facts and rules, AnsProlog has a construct called a choice rule. Choice rules give AnsProlog the ability to generate *multiple* consistent worlds. Instead of querying for a particular fact like standard Prolog, the output of an AnsProlog program is every single set of logically consistent facts that can be derived. These are called answer sets. AnsProlog has been found to be a natural fit for procedurally generating game artifacts. It is a productive language to use because a game can be thought of as objects that obey rules. Because ASP generates every possible answer set, a single program may generate hundreds or thousands of useful artifacts - a huge boon to productivity.

A shortcoming of AnsProlog is that it does not lend itself to test-driven development. To get a similar experience, the authors of *A Pragmatic Programmer's Guide to Answer Set Programming* (Cliff & DeVos, 2009) suggest building a visual feedback loop for artifacts generated by ASP. To that end, instead of writing test mocks and a test suite, it is often easier to write a rendering engine. Fortunately, the Swappy client I created accepts ASCII art representations of levels as input and, as a result, developing the renderer was easy.

Despite shortcomings, AnsProlog is a productive language for PCG. *Procedural Content Generation in Games* (Shaker et al., 2016), a book that aims to survey the entire space of PCG, contains a chapter detailing its use. There are several authors who appear to be prolific using ASP (Cayli et al., 2007,;Smith et al., 2013). The POEM Lab at NCSU, lead by Dr. Chris Martens has been productive with AnsProlog for procedurally generating content (Martens, 2018). Dr. Martens' tutorial on AnsProlog was the first tutorial I used to familiarize myself with the language (Martens, 2017). At the Strangeloop 2018 conference in St. Louis Missouri, I met Dr. Martens and was able to consult with her about my project. After demoing Swappy, she immediately felt AnsProlog would be useful, and pointed me in the direction of some popular AnsProlog works (Smith & Mateas, 2011; Smith et al. 2013). The virtues of ASP for PCG are being explored heavily in academia, and in my research, I was unable to find a commercial game using the technique.

A useful pattern for PCG with ASP is a pattern from *Procedural Content Generation in Games* (Shaker et al., 2016) I have come to call "core-sim-style". In this pattern, three programs are developed: a core; a simulation; and a style program. Both core and sim follow the define-generate-test pattern found in *Clingo Users Guide* (Gebser et al., 2014). The core is used to generate the game pieces and explain their relationships. The sim program acts as a simulation of gameplay. This artifact is not used as part of the level that can be rendered. The existence of a consistent simulation ensures that gameplay rules are followed, and that following the rules can produce a solvable artifact. Lastly, the style program is concerned with generated artifacts meeting some criteria of aesthetic standard.

Program Requirements

Intermediate and challenging Swappy levels that are manually authored can take anywhere from 8 to 20 hours to create. Even the most trivial levels take minimally around a half hour. Any program that can best this rate would be an improvement to the current process. To aim a little bit higher, we should consider that manually created levels take anywhere from a minute to fifteen minutes to solve for most playtesters. In an ideal world, the level generator could keep a steady feed of increasingly complex levels for any play session. With this in mind, the aimed-for generation time constraints should be on the order of one to fifteen minutes.

Utilizing the "core-sim-style" approach found in *Procedural Content Generation in Games* (Shaker et al., 2016), I was able to develop a level generator for Swappy in AnsProlog. The AnsProlog is run by a small Python command line application that allows the user to alter the parameters of the output level. The alterable parameters are the width of the level, and the number of character tokens to be used for the generated level. What is rendered to the user is an ASCII representation of a Swappy level. This level can be placed in a Swappy client to see and play the generated level.

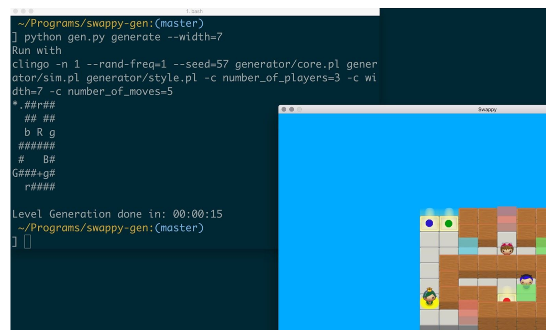

Figure 2. ASCII representation and rendered level side by side.

Implementation

I developed the generator in a combination of Python and AnsProlog. The generator consists of four parts; a command line application, and three AnsProlog programs. I built the command line application in Python using Google's *Python Fire* (Bieber & Patel, 2018). *Python Fire* is familiar to me and I find it useful for rapidly developing command line applications. The command line application shells out to run the three AnsProlog files using the grounder/solver tool Clingo. Even though the program is generating potentially hundreds of thousands of answer sets, only one is randomly chosen. The facts produced in the chosen answer set are given back to the Python utility to be parsed and rendered as ASCII art in a terminal. The output then becomes input to a Swappy game client.

A useful analogy for thinking about "core-sim-style" is to imagine a board game. Core is responsible for making the tangible things in the box. Simulation can be thought of as the instruction manual for the game. Its use is not required during gameplay, but the rules and relationships among the game pieces in the box is explained in it. Following the instructions will ensure that players have correctly played the game and explain when a player has won. Lastly, the style component can be thought of as the painting on the pieces, the illustrations on the tiles, and all of the work that goes into making the game feel and look fun.

Following along with the source repository, let's examine the code (Appendix A). (To follow along with the code, view generator/core.pl.) Lines one through three allow user input for constants, with

some sane defaults. Line seven uses the number of players in a cardinality constraint rule (see Appendix C). The rule states that an answer set must contain a set of colors with "number_of_players" elements in it. We then use the selected colors to make the rest of our game objects. Lines four to 11 are facts instantiating the entities found in Swappy. For example, if blue and green were chosen these rules would create answer sets that contain blue and green doors, players, and goals. Lines 14-17 are cardinality constraints for generating sprites. A sprite is an abstraction for what will eventually be given back to Python for rendering. A sprite fact contains a tile with X and Y coordinates, a sprite name, and a color. Each of these lines can be read as, "for every tile, maybe place this sprite here." This is done for doors, walls, players, and goals.

At this point, the facts are expanded and we could potentially render. However, the results may contain absolute nonsense. We may have answer sets where every goal, player, door, and wall are on the exact same tile. To ensure sane output, we need to constrain what is generated using constraint rules. In AnsProlog, a rule without a head can be read as "Discard answer sets if the following happens." With that in mind lines 20-43 are responsible for defining qualities of answer sets that can be discarded. Comments above each constraint illuminate their specific purpose. Some examples include, "No duplicate players," "Players do not start on goals," and "Don't put players or goals on top of doors."

Next, we examine the code in generator/sim.pl. This code is responsible for creating facts and relationships about valid ways of playing the game. In the code, the idea of a character being able to reach a tile is captured in a touch fact. A touch fact contains a color and a tile. If the green player touches the tile at coordinate (4, 5), the fact touch((4, 5), green) will be generated. We begin by having a touch generated at the location of a player sprite on line two. This initial touch will be important once we establish adjacency rules.

In Swappy, characters can walk to adjacent spaces. We define what adjacency is in lines five to eight by using some simple grid math. To define player movement via walking, we will use this adjacency definition on line twelve. The rule can be read as, "If there is a touch on tile T1, then for every adjacent tile T2, there is also a touch for tile T2." Using this definition and providing an initial touch per player, will give us a flood fill of adjacency from where the player starts.

Next, we recall that Swappy tokens are able to swap places when they are collinear. The rules in lines 15-27 allow for this. The conditions for swapping to another tile are; another character token has touched it, and either the X coordinates or Y coordinates are equal. The first rule takes care of horizontally collinear tokens, the second allows for vertical collinearity. Now that we have generated facts about what tiles players can touch, we should constrain our program to follow the rules of Swappy. To prevent players walking through walls and wrong-colored doors, we constrain movement on lines 30 and 35. The constraints discard levels where players traverse (touch) a wall or a wrong-colored door. With movement rules generated and constrained, we can now define the win conditions of Swappy. On lines 42 and 45, we say that a level can be considered complete if a player touches their goal. We also enforce that this must be done for every character token.

Finally, we want to generate levels such that each character token starts in a region with a goal in it. This exists because of a discovery during development where I found that levels without this feature are unsolvable. We define sight similar to walking, but do not constrain on doors. For a visual understanding, note that the blue character in *Figure 1* can "see" the red goal in the room adjacent to him, even though he could not walk to it. The green and red characters in *Figure 1* can also "see" the blue and green goals located in the upper right corner.

We wrap up our examination by looking at generator/style.pl to see what aesthetic properties we define. On line four we generate walls in clusters of three by enforcing a cardinality constraint on adjacent walls. In practice, this made generated levels have walls that clump up less. On line seven we have a similar constraint around the tiles adjacent to a door. This rule was intended to make sure that doors were more likely to be placed in hallways. Finally, we reject answer sets that have doors adjacent to each other. This

rule has been broken in manually created levels, but for the generator, its inclusion has assured levels look more spaced out, and generally leads to more movement required of character tokens to reach their goal.

Before settling on ASP, I took two approaches to PCG. The first approach was an agent-based approach where I started with a grid filled with walls. I would then place agents within the grid and slowly have them "carve out" playspace. The agents keep track of what they had "seen" and "done". I slowly built up rulesets and exceptions for the agents to follow. For example, "do not walk on a tile another player has," or "prefer walking in straight lines." This implementation felt like evolutionary steps towards what it might take to develop a solver for Swappy, but felt very removed from the game's rules. The agent-based solution also generated snaking paths, and the solutions were incredibly obvious. To remedy this, I would need another pass to stylize or obfuscate the level solution. I abandoned this approach because mode switching between high-level design goals, and minutiae of agent behavior felt more difficult than manual generation.

The second attempt tried a sort of back-tracking. I would start by generating a goal, and then continue to produce paths away from it. Each goal would be given a list of moves to pull from involving door placement and swapping. It was during this iteration that I began thinking about the underlying level structures and generating directed acyclic graphs out of levels. The insight of finding a directed acyclic graph within Swappy may be used in future work. This attempt was markedly more productive than the agent-based approach. However, like the previous attempt, the approach was marred with coding minutiae, and was eventually abandoned in favor of AnsProlog at the recommendation of Dr. Chris Martens.

Results, Evaluation, and Reflection

I created a level generator that can make simple Swappy levels. In some instances the generated levels stump me. The levels may be simpler than ones manually generated, but I consider this a huge personal victory. For the first time, I was able to experience a Swappy level from the perspective of a player. The insight into how *fun* playing Swappy is to players is encouraging. That said, I was unable to generate a level that needed more than a minute or two to solve. The manually generated levels may take players upwards of fifteen to twenty minutes. Looking ahead, it is clear that more rules and constraints around difficulty must be developed.

While developing the generator, I learned quite a bit about my own game. For example, the generator has a rule requiring character tokens to share a region with a goal. This idea came about during playtesting. I was playing the first successfully generated level that had rules concerning swapping, and at first glance the level appears solvable. After trying for several minutes, however, I realized the green player was locked in a region without a goal in it. This led me to critically look at all manually generated Swappy levels. I discovered that if a player is in a region without a goal in it, then the level is unsolvable. Without the generator, this discovery may not have been made.

The generator also has no issue generating patterns that I would not have otherwise seen. For example, a constraint about doors having two neighboring walls revealed that I could have doors in corners. This was always possible, but because the generator does not have instructions to make the two walls adjacent to a door collinear, it has no qualms about making corner doors. These doors allow for some interesting puzzles and, in all my time developing Swappy, I had never thought to do this. To me, the unexpected-but-useful artifacts are the most valuable, and I anticipate there is more to learn.

The generator may still produce unsolvable results. This is due to bugs around the win condition of the game. The ruleset developed states that as long as every player can reach their goal, *at all*, the level is solvable. This is not true in all cases. Consider levels where each player could visit their goal individually, which satisfies the constraint as written. However, the rules of Swappy require that all players reach their goals *simultaneously*, something the coded rules do not enforce. In order to address this bug, I researched solvability and found promising directions in *A Case Study of Expressively Constrainable Level Design Automation Tools for a Puzzle Game* (Smith et al., 2012). Smith et al. (2012) developed a directed acyclic

graph for the game *Refraction* to encode and enforce its solvability. Therefore, I generated a directed acyclic graph for Swappy levels. Such a graph can be constructed for all manually generated levels, and enforcing the existence of such a structure would result in levels that are always solvable. However, I was not able to successfully encode such requirements.

Conclusions and Future Work

Given my unfamiliarity with the problem space of PCG, AnsProlog, and ASP, a satisfactory seed for a level generation system for Swappy has been developed. However, the solution is not without faults.

The current implementation is slow, likely due to my unfamiliarity with the AnsProlog language. In some cases the generator can run much longer than the aimed-for maximum of fifteen minutes. Runs of the program also exist that may take hours, but I typically terminate these runs after the fifteen minute mark. It has been shown that scalability need not be a problem for PCG with AnsProlog (Smith & Bryson, 2014). For the current implementation, as the input parameters increase, so does the generation time (Appendix B). Exploring more efficient ways to model and run the generator may be fruitful.

Because some generated levels are unsolvable, the solution is not necessarily less error prone than I am at manually generating levels. Instead, it is *differently* error prone. I detailed some thoughts above about how I might solve this using a directed acyclic graph. In future work, I would encode a directed acyclic graph of gameplay. At this time, I don't think complete correctness is infeasible, but it remains to be developed.

At the outset of the program, I had no understanding of common patterns and practices in PCG outside of playing video games that employ it. I am now minimally conversant in AnsProlog and know quite a bit more about ASP and PCG. Also, getting practical experience with declarative programming languages may prove beneficial as the popularity of declarative infrastructure systems, like Kubernetes, continues to increase.

The generator has some vestigial code related to the number of moves each character token may make. A move is considered; walking, walking through a door, and swapping with another character. Combined with the directed acyclic graph implementation above, it may be true that this is a useful heuristic for adding complexity to a level. Future implementations should include constraints pertaining to number of moves taken.

The generator only produces a single level, and does not tie directly to a Swappy client. At present, I am manually copying output levels into another game client, developed earlier. The solution should be integrated with a Swappy client to make a seamless generation experience. Perhaps a future implementation would track the most-recently generated level constraints and gradually generate more challenging levels.

The game client used was written in Python 2, in the game framework PyGame. Future clients that integrate the solution should likely be developed in more common game frameworks like Unity or The Unreal Engine. Finally, the command line application could be transformed into a level generation web server that exposes an endpoint to consume desired parameters for generation. This would allow generator and client development to be completely separate concerns.

This solution has taken level generation for Swappy from graph paper and buttons to AnsProlog and ASP. I am impressed by the expressive power of AnsProlog. For instance, the rules that encode swapping are succinct, mathematical, and elegant. To see my game in this form was delightful. I also feel like I have another author to bounce ideas off of and take inspiration from - an invaluable asset. I believe that I have added valuable tools to my toolbelt in the form of ASP and AnsProlog. I have pushed myself into a previously unknown territory and did not come out empty handed.

Bibliography

- Bieber, D., Patel, S., Python Fire, (2018), GitHub repository, https://github.com/google/python-fire
- Noor Shaker, Julian Togelius, and Mark J. Nelson (2016). Procedural Content Generation in Games: A Textbook and an Overview of Current Research. Springer. ISBN 978-3-319-42714-0.
- Smith, A. M., & 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, A. M., Butler, E., & Popovic, Z. (2013, May). Quantifying over play: Constraining undesirable solutions in puzzle design. In FDG (pp. 221-228).
- Smith, A. M., Andersen, E., Mateas, M., & Popović, Z. (2012, May). 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 (pp. 156-163). ACM.
- Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2014). Clingo= ASP+ control: Preliminary report. arXiv preprint arXiv:1405.3694.
- Gebser, M., Kaminski, R., König, A., & Schaub, T. (2011, May). Advances in gringo series 3. In International Conference on Logic Programming and Nonmonotonic Reasoning (pp. 345-351). Springer, Berlin, Heidelberg.
- Caylı, M., Karatop, A. G., Kavlak, A. E., Kaynar, H., Türe, F., & Erdem, E. (2007). Solving challenging grid puzzles with answer set programming.
- Brain, M., Cliffe, O., & De Vos, M. (2009). A pragmatic programmer's guide to answer set programming. 49-63. Paper presented at Software Engineering for Answer Set Programming (SEA09), Potsdam, Germany.
- Smith, A. J., & Bryson, J. J. (2014). A logical approach to building dungeons: Answer set programming for hierarchical procedural content generation in roguelike games. In Proceedings of the 50th Anniversary Convention of the AISB.
- Martens, C., POEM home website, (2018) https://sites.google.com/ncsu.edu/poem/home
- Martens, C.(2017), Notes on Answer Set Programming, http://www.cs.cmu.edu/~cmartens/asp-notes.pdf

Appendices

Appendix A

Swappy Level Generator Source Code https://github.com/DanLindeman/swappy-gen

Appendix B

Number of players	Level Width	Time to Generate	ASCII Output
2	5	0.030s	+G## ## ## ##g g## .R##
3	6	12.061s	* b### G ## # #-Y### #bBb.# ######
3	7	39.820s	r### *## # Gr## ## ##Br#r ##+b## # ### ### R.
3	8	260.069s	######## #b b r # #Rb##### #.## # # #r#b## #B# ## ### *+G

Appendix C

Example Rule	**Type and Explanation**
wall(none, wall).	A Fact. A wall is a tuple of none (no color), and the sprite name ‘wall’.
door(C, door) :- color(C).	A Rule. If there is a color for color C, then there is a door of color C.
{ touch(T2, C) : adj(T1,T2) } :- touch(T1, C).	A Choice Rule. If there is a Touch on tile T1, then for every adjacent tile T2, there is also a Touch for tile T2.
0 { sprite(T, S, C) : goal(C, S) } 1 :- tile(T).	A Cardinality Constraint. If there is a tile T, for every goal of color C and sprite S, generate 0 to 1 sprites on tile T, of sprite name goal, for color C.
:- sprite(T1, goal, C1), sprite(T2, goal, C2), C1==C2, T1!=T2.	A Constraint. Discard any answer set that has two goal sprites that share a color.