Mario Level Generator using Answer Set Programming

Sanket Goutam, Varun Garg
Department of Computer Science,
North Carolina State University

Abstract

Procedurally generated content has been a great area of interest among designers, artists, musicians, and game developers. Answer-set programming, an attractive logic programming system for PCG, offers a simple and powerful modelling language to describe combinatorial problems as logic programs. Recent work has demonstrated new ASP solvers designed to better fit the run-time resource constraints of modern games. We implement two different ASP solvers to generate playable Mario levels and evaluate them based on various metrics that best portray the gameplay. We demonstrate a method through which one can concisely capture the design space of Mario as a ASP problem, rapidly create generators for different levels, and create different kinds of levels while at runtime. We compare 50 generated levels, made using Clingo and CatSAT, using both quantitative and qualitative measures focusing on users' experience and ease of use for developers.

Keywords: Answer Set programming, Mario, Level Generation, Clingo, CatSAT

Introduction

Procedurally generated content is an area of great interest. It can be employed in several domains such as music, imagery and games. There are several games based on PCG. One of most popular games "Mario" has also received a taste of PCG where new levels are generated with different designs.

We experiment with one such PCG method, Answer Set Programming, to test the level generation at runtime for Mario games. The aim of our project is to generate levels using two different ASP solvers, and present our analysis of the results that best portray a users' experience and ease of use for game developers. By concisely capturing the design space as an answer set program, we can rapidly define and expressively sculpt new generators for game levels. The ASP system will be tested using expressive range analysis as well as human playthrough of hand picked levels for both a quantitative and a qualitative analysis.

Our work focuses on the use of ASP and evaluation of the same using a method that gives the range of expressions obtainable from the generator. (Smith and Mateas 2011) provides a very good overview of the working of ASP. (Smith and Whitehead 2010) introduces the concept of testing and evaluating for the expressive range of a generator as opposed to highlighting some of its samples for demonstrating the generator's features. We aim to utilize the concepts discussed in these works and implement them for comparison using Mario.

Concerning the implementation of our generators, we capture the Mario level map as a problem set represented in both Clingo and CatSAT. After the level generation, we use Mario AI Engine 2011 (Karakovskiy and Togelius 2011). We also design specific metrics for evaluation of using these generators for playable levels. Our metrics aim to measure the generated content both quantitatively and qualitatively, and are designed keeping the users' playability and programming ease in mind.

The rest of the paper is organized as follows. In section 2, we provide a brief background of the work already done in this domain, with focus on different PCG problems, ASP solvers, and current implementations of level generator for Mario. Section 3 describes our approach for the level generation including the methodology for capturing the design space of Mario as a ASP problem. We also describe implementation logic that are common to both Clingo and CatSAT. Section 4 presents our evaluation metrics, followed by an analysis of all the levels generated, expressive range analysis of 50 unique levels, and the time taken by each generator. In section 5, we discuss our results and provide an in-depth comparison of the performance of both generators, and future work in extending ASP for game level generation.

Background

Procedural Content Generation (PCG) is a game-design technique through which developers create game content via an automated process rather than via hand-authoring. It readily provides the means to generate entire design spaces, and game levels on the fly. PCG typically excels at creating a large number of levels in a short period of time.

Smith et al. discuss the concept of PCG-based game design as a way to create new kinds of playable experiences. The authors analyze the different ways PCG is used in games and the impact it has on the player's experience. Their analysis of the PCG-based games considers evaluation on the following measures:

- Replayability and Adaptability, or the ability of the PCG to adjust content in reaction to player actions or skill levels

- Game Mechanics and Dynamics, or the ability of the PCG systems to augment traditional mechanics or augment new dynamics entirely
- Player Control over Content

They provide an experimental PCG-based game that expresses the idea of creation through exploration, and successfully highlight the use of PCG to provide a new kind of playable experience. Their experiments with building Rathenn has shown the highly iterative process of PCG based game design, with the game pushing the constraints on the generative system.

However, a generative procedure may fail by producing an undesirable artifact such as an unsolvable puzzle, a nonsensical story, or an inexpressive and monotonic level. As claimed by (Smith and Mateas 2011), PCG is concerned with two design problems: The first is a concrete design problem dealing with the production of game content with desirable properties. The second is a meta-level problem dealing with the production of generative procedures with desirable properties (such as, among others, the ability to produce desirable content). Answer set programming has emerged as a declarative programming paradigm with particularly potent affordances for describing the design spaces of PCG problems. Although nominally designed for knowledge representation and search-intensive reasoning tasks, it is easily re-purposed for answer set synthesis in which ASP is exploited primarily for its generative capabilities.

We explore the capabilities of Answer Set Programming for level generation in 'Mario' and evaluate the design space thus generated based on metrics that capture the differences from other generation methods. Prior work (Shaker et al. 2012) has shown implementation of evolutionary algorithm with grammatical representation to generate content for Super Mario Bros, as it allows exploration on a wide space of possibilities. The effectiveness of using an evolutionary search algorithm for content generation, however, is limited by the expressiveness of the design grammar. (Snodgrass and Ontañón 2014) describe a method of procedurally generating maps using Markov chains. Their implementation learns the statistical patterns from human-authored maps, which are assumed to be of high quality. They then use the learned patterns to generate new maps. While this method is largely successful in generating good Mario levels, their model largely depends on prior hand crafted maps.

Our contribution, through this paper, is to fill the research gap in level generation with no prior learning, for games like Mario. By allowing a programmer maximum control over the design space of levels we can ensure that more unique and expressive levels are generated. We provide a working framework for Mario level generation using ASP that allows further research on building online level generators. We also evaluate the expressiveness of the content generator that would allow the developers to correlate difficulty levels with the engagement of the players and thus provides more developer control over the content.

Methodology

In this section, we describe how we capture the the Mario levels design space as a ASP problem, the constraints that we define on the problem, and how the generated levels are encoded into the Mario game engine.

Representing the design space

We represent each level map as an $w * h$ two-dimensional array *level-map*, where h is the height of the map, and w is the width. Each cell *level-map(i,j)* corresponds to a tile in the map, and can take one of a finite set of *tile-types*, which represent the different types of tiles that we can represent in the game. In ASP, each of these tile types becomes a standalone proposition. In general, the representation of each tile type in the game depends on the developer. By allowing a developer define all the various tile designs we ensure that they have maximum control on the design space.

For our implementation, we have incorporated a total of 17 tile types or designs. These include the basic tile types of "Coin block", "Brick block", "Air", "Land", "Coins", "Goombas", "Koopas", "Pipes", "Gaps", "Power-up blocks" and also include specific design patterns that we created by combination of different block types. This allowed us to represent a more diverse range of design in our generated levels. The figure below shows one such generated level where a combination of design patters (coin blocks) defined by us is used to create an interesting pattern.

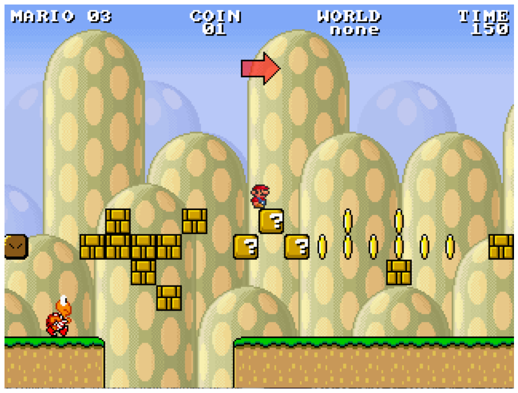

Figure 1: Mario Level showing unique design patterns

Constraints on the generators

Every constraint is designed to ensure a level's playability. Constraints are applied on the various tile types. There are two types of constraints:- position based constraints and count based constraints.

Position based constraints: The position based constraints ensure that the levels are logically sound in terms of gap placement, pipe placement and placing the ground. These constraints are crucial in ensuring playability as well as getting the basic layout of a Mario game.

- Ground constraint: - This constraint ensures that there is no random ground placement in the air and that the ground can cover only the lower most space of the level.

- Gap Placement:- The gap placement constraint plays the role of ensuring a level is playable and prevents placement of gaps adjacent to each other. The length of adjacent gaps can go beyond the range of Mario's jump and therefore render the level unplayable. Another position constraint ensures that there is no gap in the first few blocks of the level, so that the player can at least have a start.
- Pipe Placement:- This constraint removes those levels where the pipes are too tall and where the pipes are directly above a gap thereby floating in the air.

Count based constraints: The count based constraints ensure that none of the special tile types (impassable tiles) cover a majority portion of the map and block Mario's path. Also these constraints remove the possibility of having levels with many enemies or few enemies. These constraints affect the difficulty of the game by governing coins, bricks, pipes, enemies, jumps and power-ups.

- Coin Block constraints:- We restricted the number of coin block patterns in a given level. The number of coin blocks of a given pattern can be between 1 to 3. There are 2 different patterns and therefore a maximum of 6 coin block patterns are present in a game. The first pattern has one coin block and the second pattern has three coin blocks.
- Coin constraints:- Here there are 3 distinct patterns and each of those patterns can occur in the game once or twice. The 3 patterns have 1, 3 and 5 coins. Together with coin block patterns there can be a maximum of 30 coins in a level. We chose such constraints so that every 4 levels which is a categorized as a world in the Mario games, the player gets a life (100 coins make a life).
- Brick constraints:- The number of brick patterns in a level can scale from 3 to 6. There are three patterns and each pattern can occur at most two times in a level. When patterns appear at the same spot or overlap, they create new shapes. One such example is given in figure 1.
- Enemy constraints:- The number of Goombas (sentient mushrooms) in any level ranges from 3 to 6. The number of Koopas (turtles) in any level ranges from 1 to 3.
- Pipe constraints:- The number of pipes is restricted to a minimum of 3 pipes and a maximum of 6 pipes. In the current scenario, pipes are just for aesthetic purposes and to add jumps to make the gameplay varied. These pipes can also be used a placeholder for flower enemy, however they haven't been added to the generator yet.
- Jump constraints:- The number of jumps in a level ranges from 3 to 6. There are three sizes of jumps in a level, 2 block wide, 3 block wide and 5 block wide. Each jump length can be present in a level at most 2 times. The jump count and position constraints ensures that there ample ground for mario to run on as well as Mario can make all the jumps.
- Power-up constraints:- There at most 2 power-ups in a level. If Mario is large, the game engine places a Fire-Flower in the power-up block and if Mario is small there will be a mushroom there.

Encoding generated levels into the game engine

We use the Mario AI 2011 (Karakovskiy and Togelius 2011) software as out game engine as it afforded freedom and ease of use while encoding the levels.

The output of both CatSAT and Clingo is a list of 3-tuples (X,Y,T) where X and Y are position co-ordinates and T is the tile type. Since the position co-ordinates X and Y are unique, each tuple is unique. These tuples are fed to the engine. The engine creates a level object and populates the map of the level with the various tile values. Whenever a special tile pattern is encountered, the adjacent tiles have their values changed according to the pattern. However, if two patterns are next to each other such that there is overlap, then the pattern with the lower X co-ordinate wins and has its pattern written over the losing pattern. This creates new patterns and designs.

Enemies are placed at the location indicated by their tuple and the engine simulates their behaviour and movement. Pipes heads are placed at the location present in their tuple and the stem extends till the ground. Since, there is no check for enemy location and pipe location overlap, there are levels where an enemy is stuck in a pipe and keeps moving to and fro within a pipe as the engine makes the enemies go in the opposite direction upon collision with a pipe or impassable object.

Since, we are generating levels of width 100, the final flag is placed at X coordinate 100. Upon launching the level, the engine parses a text file and creates the level and displays a GUI for playing the game.

Metrics

In order to best demonstrate the range of content that can be generated using ASP solvers for games like Mario, we considered two main aspects of game design. The first set criteria are user-based and focuses on game play. The metrics that we considered under this criteria are playability of the level which tells if the generated level is playable or not, and the difficulty of the level. We note that the difficulty of a level is more of a qualitative measure, and thus we use three different quantitative metrics to represent difficulty of each level. These metrics are the time taken to finish each level, number of actions taken to finish, and time spent by Mario in the air (jumps performed).

The second set of criteria which we defined were focused on level design, and are evaluation metrics which are more focused for game developers. The metrics defined under this criteria include the spreadness of each level, and time taken for level generation. Spreadness is a measure for how varied the each generated level is. As a game designer using a generative model to create game levels, it is quite important that the generated content has some variation. For instance, we cannot have a generated level where there are no obstacles and Mario can just run straight till the end. Conversely we cannot also generate levels where the map is filled with random obstacles making it impossible for the user to proceed. A sweet spot between relatively sparse and excessively filled level is desired.

The second metric which is the time taken for generation of each level is one of the most important metric. This is because we want the levels to be generated online, and having a low latency for the generator to create new levels is very important. We record the time both the generators take to solve the ASP problem, and not the complete work flow of the system. This is because the usage of Clingo and CatSAT for level generation is quite different, and was not applicable to create a common complete work flow.

A summary of all the criteria is presented in Table 1.

Results

We tested 25 levels each for both CatSAT and Clingo. Each of the level was played by an AI. The AI we used was the winner of the 2009 Mario AI gameplay track created by Robin Baumgarten. It is based on the AStar algorithm and its sole aim is to win the level with the leaset possible time. We used software from the 2009 Mario AI gameplay track. This software has its own Mario engine that is similar to the 2011 track as well as the means to add code to record other metrics.

Playability

The first result we present is the number of playable levels generated. This basically refers to whether Mario can reach the end and touch the flag. The results are given in Table 2.

From table 2, we can see that both generators are creating a high percentage of playable levels. The unplayable levels occur due to the placement of a pipe with a special block and so mario cannot jump over the combination of the two. Also, a level was unplayable because an enemy goomba spawned at the exact same location of Mario's spawn. This results in Mario's death the moment the level starts.

Difficulty

Difficulty is a subjective score and hence we try to make it based on a total of three metrics. Time taken, actions taken and jumps. Since, the AI focuses to finish the level in as little time as possible, having lower values for all three metrics denotes to some extent that the level is easy.

The playtime for both Clingo and CatSAT is shown by the graph in figure 2. The mean time and standard deviation are given in table 3. Unplayable levels do not have data points as data is recorded for levels that are completed. In general, levels generated by Clingo take slightly more time than levels generated by in CatSAT. This is attributed to both a lower number of enemies and jumps present in levels generated in CatSAT. The graph shows one outlier in Clingo which goes above 40 seconds. This level was special as it required Mario to backtrack and jump to land on a brick to gain a height advantage. The AI was unable to clear this level, and these figures were obtained via a manual play through. This level can therefore be categorized as a difficult level.

Another means of gauging the difficulty of the level is by the means of number of actions performed. Actions here refers to jumps, movement and sprint. Higher the number of actions, higher is the expected difficulty of the level. Since the AI is motivated by taking the least time to complete the level, it tends to use the movement to the right and sprint as its primary actions. Any other actions such as jumps, going back, stop sprinting and start sprinting again relates to the level having elements that need attention. The presence of a gap warrants a jump, similarly the presence of two closely placed gaps requires a slowdown to precisely land the jump. The values for actions taken is given by the graph in figure 3.

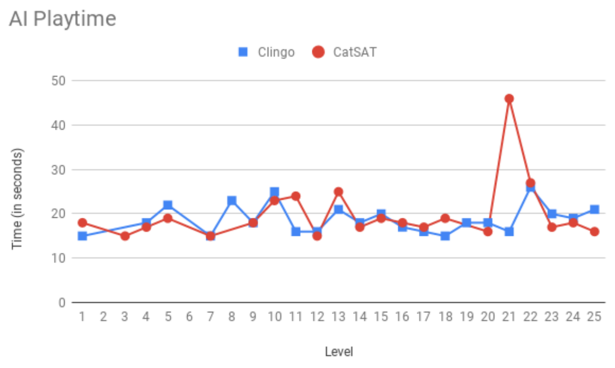

Figure 2: Mario AI Playtime

Once again level 22 the outlier in time, is an outlier here. Since, it requires backtracking and precise jumps the number of actions required shoots up to 620. The mean and standard deviation of actions required is given in table 4. We can see that Clingo is creating more varied levels with a higher standard deviation and also more difficult levels with a higher mean actions required per level.

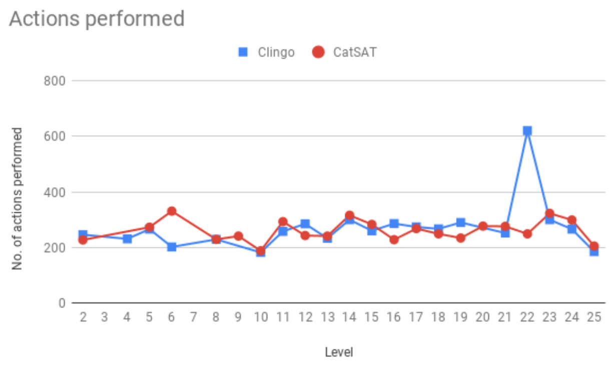

Figure 3: Actions required to pass the level

The last metric for evaluating difficulty is Jumps. This metric records the number of actions for which Mario is in the air. Here, Mario whenever Mario jumps, this metric increases once for frame till Mario touches the ground again. The values for jump/ in air for every level is given in figure 4.

We can see that for Clingo Mario is in the air for a larger number of frames. This is because of the presence of more enemies and gaps. Also again level 22 is the outlier, due to presence of the outlier that required for multiple attempts. In

Table 1: Summary of evaluation metrics

Metric	Criteria	Description
Playability	User centred	Denotes if the level is playable
Time taken	Difficulty criteria	Represents the time taken by the AI to finish the level
Actions taken	Difficulty criteria	Represents the number of actions taken by the AI
Jumps taken	Difficulty criteria	Represents jumps (or the time spent by Mario in air)
Spreadness	Developer centered	Represents how many unique tiles exist in the level
Time Analysis	Developer centered	Time taken for level generation

Table 2: Playablility Score of Generators

Solver	of playable levels	% of levels playable
Clingo	22/25	88%
CatSAT	21/25	84%

Table 3: Time taken by AI

Solver	Mean time(seconds)	Standard Deviation
Clingo	19.95	6.84
CatSAT	18.7	3.17

Table 4: Actions required to pass the level

Solver	Actions required	Standard Deviation
Clingo	270.7	87.05
CatSAT	263.5	39.7

general, Clingo has more levels that have more jump frames compared to CatSAT.

The mean and standard deviation for jump values is given in table 5. Here CatSAT has a slightly higher mean but once again Clingo is more varied.

Table 5: Number of jump frames

Solver	Jump Frames(mean)	Standard Deviation
Clingo	49.04	29.14
CatSAT	49.95	17.25

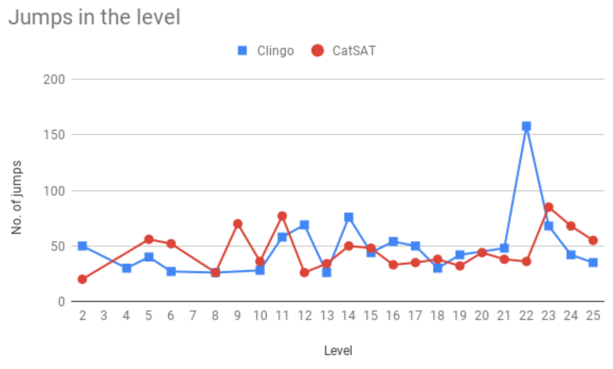

Figure 4: Amount of time Mario is in the air

Spreadness

We define spreadness of each level as the number of different tile types that are generated within each zone of the level. The base logic behind introducing this metric was that the visible area of the map at any time is only 20-25 blocks wide. So we wanted to ensure that the generated levels in each visible range was expressive enough to make the level interesting, and should not be monotonous as the player progresses through the level. We divided the generated levels into 5 consecutive sections for expressive range analysis and evaluated the spread for each section.

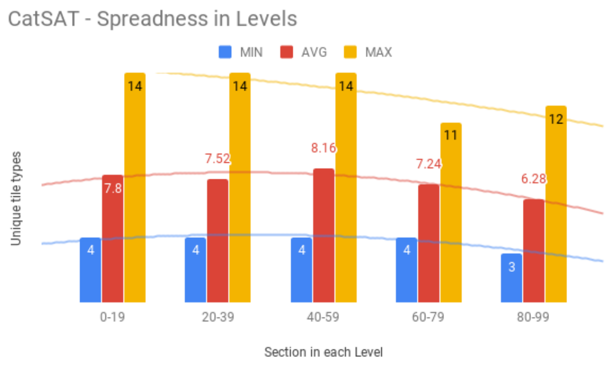

Figure 5: CatSAT - Spreadness analysis

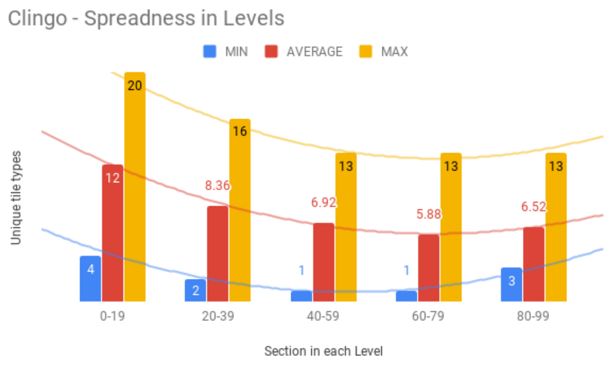

Figure 6: Clingo Spread analysis

As can be seen in the figures above, CatSAT performs much better in providing more spread out levels in comparison to Clingo. The levels generated by Clingo tend to have localized spread, and do not really maintain the unique spread throughout.

Time Analysis for level generation

One of the most important criteria among all the others is the time it takes for the generator to generate each level. We recorded the time for all 50 levels and the statistics are shown in figure 6.

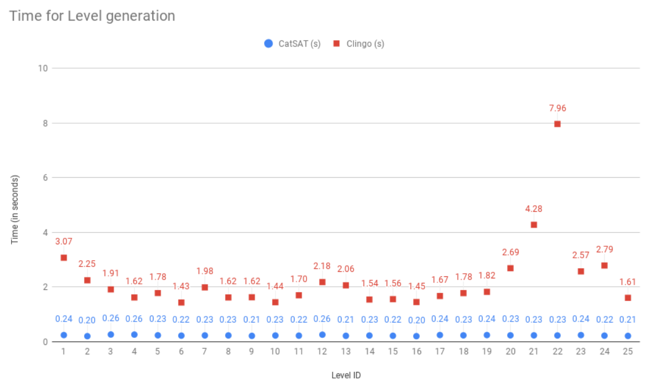

Figure 7: Time for level generation

As it can be clearly noticed from the graph, CatSAT performs much better in generating levels within a fraction of a second. Clingo however takes much longer to generate each level, typically in the range of a few seconds.

Discussion and Future Work

Based on the several evaluation criteria that we have covered above, we conclude with the following discussion about the evaluation of two generators for online level generation.

- In terms of programmer specific criteria, CatSAT provides much more flexibility of programming and ease of use as compared to Clingo. The ease of representing propositional logic in CatSAT ensures that it can be easily integrated into any game.
- The runtime of CatSAT to generate levels is also a huge plus towards programmability because it means that it would make generating online levels during playtime much more practical as compared to Clingo.
- Based on the user specific evaluation criteria, we conclude that Clingo performs better in the aspect of generating more interesting levels than CatSAT. When we manually analyzed the various levels generated, we found Clingo generating many interesting levels which in one case was very similar to the original Mario levels.
- The difficulty measures evaluated for both the generators also demonstrate that Clingo on average generates more difficult and varied levels than CatSAT. This is verified from the several difficulty metrics that we have measured which show that Clingo achieves a higher mean and variance on the quantitative measures.

Based on the findings summarized above, we have successfully demonstrated the application of ASP solvers for level generation of a game like Mario. The various metric evaluation also underpin our hypothesis that using ASP solvers as the generation method for online level generation provides a promising area of research. We have demonstrated the applicability of ASP solvers but still a research gap that needs to be addressed lies in the various ways of expressing game level space as ASP problems. We realize that this might be a difficult challenge for most games to represent their design space in terms of propositions. But the games for which it can be done, using ASP to generate unique, and previously unseen levels from scratch can be done using ASP solvers.

References

Karakovskiy, S., and Togelius, J. 2011. Mario AI Competition. http://julian.togelius.com/Togelius2013The.pdf.

Shaker, N.; Yannakakis, G. N.; Togelius, J.; Nicolau, M.; and O'Neill, M. 2012. Evolving personalized content for super mario bros using grammatical evolution. In *Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'12, 75–80. AAAI Press.

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. 4:1–4:7.

Smith, G.; Othenin-Girard, A.; Whitehead, J.; and Wardrip-Fruin, N. 2012. Pcg-based game design: Creating endless web. In *Proceedings of the International Conference on the Foundations of Digital Games*, FDG '12, 188–195. New York, NY, USA: ACM.

Snodgrass, S., and Ontañón, S. 2014. Experiments in map generation using markov chains. In *FDG*.