Precomputing Player Movement in Platformers for Level Generation with Reachability Constraints

Vivian Lee
Northeastern University
lee.viv@northeastern.edu

Nathan Partlan
Northeastern University
partlan.n@northeastern.edu

Seth Cooper
Northeastern University
se.cooper@northeastern.edu

Abstract

Procedural content generation via Machine Learning (PCGML) creates game levels from examples. PCGML for platformers with physics-based player movement generally has not guaranteed the reachability of goals, relying on post-generation filtering approaches or game-specific heuristic rules. In contrast, constraint-based PCGML can provide gameplay guarantees, but has typically been applied to games with simple grid-based movement. In this work, we present a constraint-based PCGML approach for platformers with physics-based player movement that guarantees playability and provides design controllability. Our approach exhaustively precomputes all possible player movement states in example tile-based platformer levels. It extracts metatiles containing local state information and uses them in constraint-based level generation that ensures legal tile neighbors and consistent movement state transitions. The approach can ensure constraints on gameplay, such as the reachability of goals, guaranteeing level playability, and other elements like platforms and bonuses. It can also incorporate other designer-controllable constraints.

Introduction

Procedural content generation via Machine Learning (PCGML) is an approach to game level generation that learns from existing levels (Summerville et al. 2018). By learning to generate unique levels from hand-designed ones, PCGML methods can complement and support human designers' creativity. They can generate new ideas that match well with the designer's style and needs, helping with brainstorming or refinement of a level design.

While researchers have developed PCGML approaches to generate levels for platformer games, these systems typically do not provide guarantees about the reachability of level elements, such as goals, and thus whether the resulting levels are playable. They often rely on post-generation filtering and playability heuristics and start over if the level is not playable (Snodgrass and Ontañón 2016). Constraint-based PCG approaches can incorporate reachability constraints that guarantee playability, but have typically been applied to games with simple tile-based movement rules (Nelson and Smith 2016).

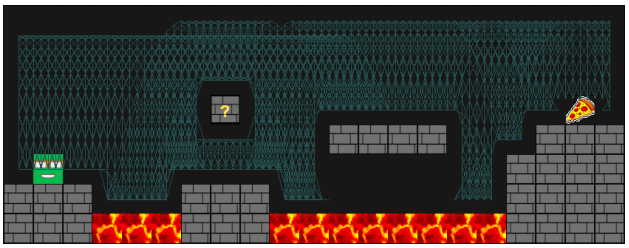

Figure 1: Screenshot of *Turtle Loves Pizza*, showing the turtle, blocks, hazards, a bonus, and the pizza goal. Positions and transitions of enumerated state graph overlaid in teal. Border tiles omitted for clarity.

Beyond playability, PCGML methods often struggle to guarantee other attributes of the resulting levels such as the distribution of design elements or the reachability of collectible items. Many PCGML methods rely on neural networks, probabilistic graphical models, and other similar approaches that are not easily interpretable or controllable. Designers may need to iteratively re-train, often guessing what might help the system achieve their desired vision (Summerville, Philip, and Mateas 2015). Constraint-based methods allow designers to write a declarative description of their requirements for the resulting level.

In this paper, we propose an approach to constraint-based PCGML for platformers with physics-based player movement that allows reachability and other designer-controllable constraints. Our approach precomputes all possible player states, based on the game's physics and movement rules, in the training levels, and associates the tiles with those states and the transitions between them to create *metatiles*. By tracking and constraining state transitions when generating levels, we ensure reachability of specific tiles like goals and collectibles. We also include tile neighbor constraints inspired by Wave Function Collapse (WFC) (Gumin 2016), and generate levels by solving the constraints using Answer Set Programming (ASP) (Gebser et al. 2011). We show how this can be combined with other constraints to ensure other design requirements, e.g. specific ranges and numbers of tile types. This process can be divided into 3 steps: (1) enumerating the state graph, (2) extracting the metatiles and constraints and (3) generating a new level that satisfies all the

constraints. We consider this to be a type of PCGML as some of the constraints are learned from the training levels.

We applied this level generation technique to a simple tile-based platformer game we created called *Turtle Loves Pizza* (TLP), shown in Figure 1. We trained on simplified versions of *Super Mario Bros.* levels from the Video Game Level Corpus (VGLC) (Summerville et al. 2016).

We applied two types of reachability constraints: *playability*, whether the generated level contains a path from the start to each goal; and *usefulness*, whether platforms and collectibles in the generated level are reachable and can be used or collected. The ideal generated level should be both useful and playable. We show that our approach can be used to generate playable levels that also meet additional user-specified design constraints including specific level dimensions and the number of certain tile types in the generated level.

The focus of this work is to propose a new technique for generating novel levels from existing ones in platformers. The contributions are (1) a level generation approach which ensures reachability and allows for other controllable constraints using constraint-based PCGML and precomputed player movement and (2) a demonstration of the approach using levels from an established domain with movement rules from a custom platformer game.

Related Work

PCG for Platformers Many researchers have proposed and tested methods for generating levels for platformer games. One line of work used designer-defined libraries or grammars, often paired with constraints or optimizations. Compton and Mateas (2006) proposed hill-climbing to generate segments of levels by difficulty, stitched together with grammars. Inspired by Spelunky (Yu and Hull 2009), Mawhorter and Mateas (2010) created more flexible levels, at the expense of playability guarantees, by anchoring hand-defined chunks based on player movement. G. Smith et al. (2010; 2011) followed these concepts of pattern analysis, adding the idea of rhythm and "beats," by using a constraint solver and reactive planning. Another approach employed graph grammars (Londoño and Missura 2015). Though some of these techniques guaranteed playability, they only supported a single path through each level, and required manual work to define the available patterns.

Seeking to learn such patterns from training data, Sorenson and Pasquier (2010), Shaker et al. (2012), and Togelius and Dahlskog (2013) employed evolutionary algorithms. These did not guarantee playability, but did use fitness functions to search towards it. Controllability of the output, however, was lacking. Others tried probabilistic graphical models such as n-grams and Markov Models (Dahlskog, Togelius, and Nelson 2014; Summerville, Philip, and Mateas 2015; Snodgrass and Ontañón 2017). However, these models have difficulty respecting global patterns or constraints (Summerville and Mateas 2016). Snodgrass and Ontañón (2016) approximated A* pathfinding to check for playability, but their test-and-regenerate approach could not guarantee it.

Finally, others have focused on artificial neural network (ANN) approaches, beginning with Laskov (2009) and continuing with Hoover, Togelius, and Yannakis (2015), whose neuroevolution technique was inspired by music theory, and then by Guzdial and Riedl (2016). These, however, also did not guarantee playability. Summerville and Mateas (2016) include special path tiles, but these tile-based paths may not reflect the exact player movement rules. ANNs also lack explainability and transparency, making them less legible for designers. Recent efforts to integrate PCGML with mixed-initiative tools (Hoover, Togelius, and Yannakis 2015; Guzdial, Liao, and Riedl 2018; Guzdial et al. 2019) may help by providing feedback and integrating playability checks (Hoyt et al. 2019), but ANNs remain difficult to train and control. Our approach, using a constraint solver to generate coherent levels that respect playability constraints, affords designers relative flexibility and control to specify additional local and global constraints on levels.

Constraint-based Level Generation Outside of platformers, researchers have experimented with constraint-based level generation. In games, this began with the level design tool SketchaWorld by Smelik et al. (2010), followed by work by A. Smith et al. (2010; 2011) on generating puzzle game designs and levels using ASP. As mentioned above, Tanagra and Launchpad applied constraint solving to platformer levels (Smith, Whitehead, and Mateas 2010; Smith et al. 2011), but their approach could only generate a single player path. Horswill and Foged (2012) applied constraint solving to ensure playability in dungeon generation.

Several released games use versions of Wave Function Collapse (WFC) (Gumin 2016), a method inspired by texture synthesis and model-based synthesis (Harrison 2005; Merrell 2009). Others have noted that WFC is, essentially, constraint solving without backtracking (Karth and Smith 2017), and a PCGML method that learns from examples (Karth and Smith 2018). Using ASP to re-implement WFC in a full constraint solver has proven successful in generating playable levels for games with simple movement rules (Nelson and Smith 2016; Scurti and Verbrugge 2018). Sandhu, Chen, and McCoy (2019) further demonstrated how design constraints can be incorporated with a WFC approach to generate non-repetitive levels for a tile-based maze game.

Going beyond previous applications of constraint solving to platformers, our approach learns from existing levels to automatically determine playability constraints, melding ideas from WFC and ASP for dungeon generation with platformer physics modeling.

Precomputation and Sampling Previous work has also applied extensive or exhaustive computation of gameplay states. One application of this has been improved runtime performance. Stanton et al. (2016) extensively precomputed gameplay states to allow for high-quality rendering on mobile devices, and Stanton et al. (2014) adaptively precomputed complex fluid dynamics that could be prohibitively expensive to compute at runtime. Another application is testing and analysis. Bauer and Popović (2012) used rapidly-exploring random trees to analyze and visualize player movement in a platformer game to support level editing.

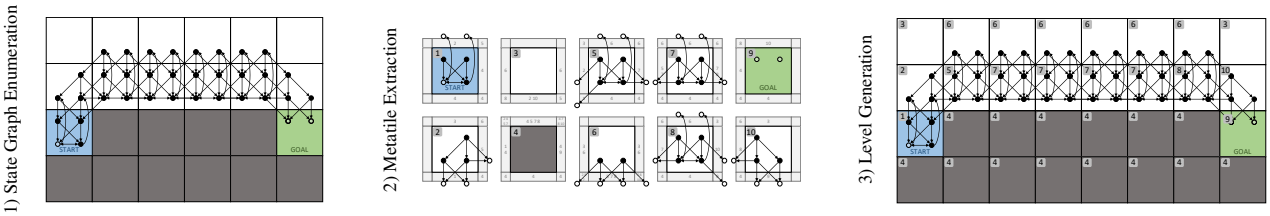

Figure 2: Illustrative example of the proposed level generation process. This simplified example does not use TLP physics; the state only consists of the (x, y) position with start and goal flags, represented by the node color. 1) Given an input level and the game's movement rules, a graph of all reachable states is enumerated. 2) Next, all unique metatiles are extracted. A metatile consists of a tile type and a state graph. We also track which metatiles were neighbors in the training level. 3) Finally, a constraint solver is used to assemble metatiles into a new level such that metatile neighbor and transition destination constraints are satisfied, along with any additional constraints supplied. Self transitions and border tiles omitted for clarity.

Overview

To generate levels using reachability constraints, we analyze existing playable levels to extract information from each tile and build a set of constraint rules that new levels must satisfy. We organize our method into: (1) enumerating the state graph, (2) extracting the metatiles and constraints, (3) solving the constraints to generate a new level. Here we discuss the generic high-level approach, summarized in Figure 2.

Input Our approach takes as input (1) the game's movement rules and (2) a playable level for training.

The game's movement rules take an existing player state and an input action and return the resulting state. The player state contains *at minimum* a position component, a flag for being the start state, and a flag for being a goal state.

Next, we define a few simplifying assumptions about the movement rules in the game. We assume that the player's movement is deterministic and that the player's terminal velocity cannot exceed the length of a square tile. Hence, we assume that the movement rules are *local*: the player's movement, current state, and next state are only affected by the player's surrounding 3x3 neighborhood of tiles at each possible state in the level. Thus, we assume that the absolute position of the state and neighborhood does not matter, only the state's relative position within its local neighborhood. These assumptions are used to extract metatiles which are then used to generate new levels (discussed below).

A valid training level is an existing level, represented as a grid of tile types, with a start tile (which defines the player's start state), at least one goal tile, and a playable path from start to goal. In this work, we assume all tiles along the outer edge of the level (and none of the interior tiles) are *border tiles*, which block the player from going out of the level. We also assume that levels themselves are static, meaning tiles will never change positions during gameplay.

State Graph Enumeration The first step in the process is to enumerate the player's state graph for the input level. We begin with the player's start state and use the game's movement rules to exhaustively precompute every reachable state in the level. The transitions in the directed graph represent the player's ability to move from a source state to a destination state by performing a particular action. The enumerated state graph for a valid input level will always contain a path from the start state to every goal state.

Metatile Extraction After we enumerate the state graph for the input level, we extract the level's *metatiles*. A metatile contains a tile type, a graph of all the player's states within that specific metatile, and the transitions between them. A metatile's state graph also includes *outgoing* edges, edges where the destination states are in adjacent metatiles. Each metatile's graph is a subgraph of the fully enumerated level state graph. Every tile in the input level has a corresponding metatile. When extracting metatiles, each state's position is offset to be relative to the metatile's location in the input level. This way, with a level's metatiles and knowledge of their locations in the input level, the metatiles' graphs can be re-joined to reconstruct the fully enumerated state graph. Each unique metatile extracted from the input level is stored and assigned a unique ID.

Once we have extracted the set of unique metatiles from the input level, we examine the input level to determine the legal adjacent neighbors for each unique metatile. These are the metatiles that can be placed in each of the 8 possible neighbor positions surrounding each metatile.

At this point we have finished analyzing the input level and have obtained (1) a defined set of metatiles, each containing a unique subgraph of player states and their transitions and (2) the learned adjacency rules for each metatile in the set.

Level Generation Finally, we input the metatiles and their adjacency rules into a constraint solver and use WFC to assemble instances of the metatiles into a new generated level.

During level generation, when a metatile is assigned to a position in the new level, all of its associated states and transitions are also placed in the level and offset to the assigned position. We use a technique similar to one suggested by Nelson and Smith (2016) to track state reachability: the start state is inherently reachable and for any source state that is reachable, all destination states that can be transitioned to from the source state are also reachable.

To assemble metatiles into a level, we use the following generic constraints:

- *Size*: Levels are rectangular and must satisfy a given width and height, measured in tiles.
- *Metatile neighbors*: Each of the 8 neighbors of a metatile

Input Level	Input Columns	Enumeration Time	Rule Gen. Time†	Other Precomp. Time	Output Size	Avg. Ground Time	Avg. Solve Time	Generated?
1-1	204	5m	7m	1m	50%	10m	5m	+
					100%	20m	11m	+
					150%	33m	20m	+
1-2*	160	4m	5m	1m	50%	7m	3m	–
					100%	14m	7m	+
					150%	22m	13m	+
1-3	152	2m	3m	1m	50%	5m	3m	–
					100%	10m	5m	+
					150%	16m	8m	+

Table 1: Summary of outcomes from size test. All levels have 17 rows; sizes include border tiles. Average times are mean for 5 levels. Rule gen time is the time it took to generate the generic ASP rules from the extracted metatiles and constraints for each input level. *SMB 1-2 did not include the *Platform reachability* constraint. †After collecting this dataset, we were able to optimize the rule generation step to just a few seconds.

Input Level	Input Blocks	Trial Blocks	Generated?	Input Hazards	Trial Hazards	Generated?	Input Bonuses	Trial Bonuses	Generated?
1-1	534	500	*+	7	1	*–	13	1	+
		750	*+		5	+		5	+
		1000	‡–		10	*+		10	+
1-2	614	500	+	16	1	–	9	1	–
		750	+		5	–		5	–
		1000	+		10	–		10	+
1-3	222	200	†–	96	50	*–	1	0	+
		300	+		75	+		5	–
		400	*–		125	–		10	–

Table 2: Summary of outcomes from controllability test. Timings were similar to those for 100% size, except for those with a * took 1.5–3x as long to solve, † took roughly 80m to solve, and ‡ were stopped after roughly 24h.

must be one of the metatiles that was seen neighboring it in the same direction in the training level. This is based on the WFC (Gumin 2016) neighbor constraints.
- *Border tiles*: The outer edge tiles are assigned to be border tiles. Interior tiles must not be border tiles. This ensures that all 8 neighbors exist for all interior tiles.
- *Transition destination*: If a source state and transition out of that state exist, the transition's destination state must also exist. This is based on the tile reachability rule from Nelson and Smith (2016). (Note that this does not require the destination state to have had a corresponding incoming transition in the training level).
- *Goal reachability*: All goal states must be reachable. This ensures playability.

We used the Potassco (Gebser et al. 2011) tools to run an ASP solver to find a placement of metatiles that satisfies the defined constraints. Other games may apply additional constraints, as we did in this work (discussed below). With this approach, we can generate new levels that satisfy defined reachability and design constraints from a small training set and elementary ML technique (Karth and Smith 2018).

Application

To explore our approach, we implemented a tile-based platformer called *Turtle Loves Pizza* (TLP). The player controls a turtle that can move left and right and jump. The turtle's (x, y) position is based on its center. The turtle moves smoothly within tiles and can occupy many possible positions within them due to the physics-based movement rules. Its goal is to traverse the level to collect the pizza at the end.

There are several tile types that can be used in the game: empty tiles that the turtle can move though, block tiles that block the turtle from moving, hazard tiles that kill the turtle when touched, bonus tiles that behave like blocks but give a score bonus the first time they are hit from below, a start tile indicating where the turtle begins in the level, and a goal tile that completes the level when touched.

The movement rules in TLP are based on simple physics rules: the turtle's state has an x and y position and an x and y velocity. For simplicity, we used integers for position and velocity. Jumping sets the y velocity to its maximum upward value, and gravity constantly accelerates the turtle downward in y. Moving left or right happens at a constant x velocity. In addition to position and velocity, each state in TLP has a flag for being a start, a goal, resting on the ground, or dead. Each state also has an indicator for being in contact with a bonus tile that can be collected in its local tile neighborhood, if any; the indicator specifies the cardinal direction of the bonus tile relative to the turtle. Thus, each state can be considered a tuple of $(xpos, ypos, xvel, yvel, isstart, isgoal, isdead, isonground, whichbonus)$.

In order to have interesting input levels, we used levels from Super Mario Bros. (SMB) from the VGLC (Summerville et al. 2016). Note that, although we used SMB levels, we used TLP player movement rules; these are different from Mario's but ensure that the levels are still playable. To prepare SMB levels for TLP we made several modifications, including mapping SMB tile types to TLP tile types; adding an additional bottom row of hazard tiles where there were pits in SMB so that the player would be classified as dead after falling into a pit; adding border tiles around the perimeter of the level; defining start and goal tiles, and making minute tile placement adjustments as needed for playability (e.g. removing a tile that Mario can pass through with a special power-up but the turtle cannot).

In addition to the generic constraints discussed above, we used the following constraints to generate levels:
- *Start and goal tiles*: There must be exactly one start tile within the first 10 columns and one goal tile within the last 10 columns.
- *Other tile counts*: The number of block, hazard, and bonus tiles must be within $\pm 20\%$ of the number in the

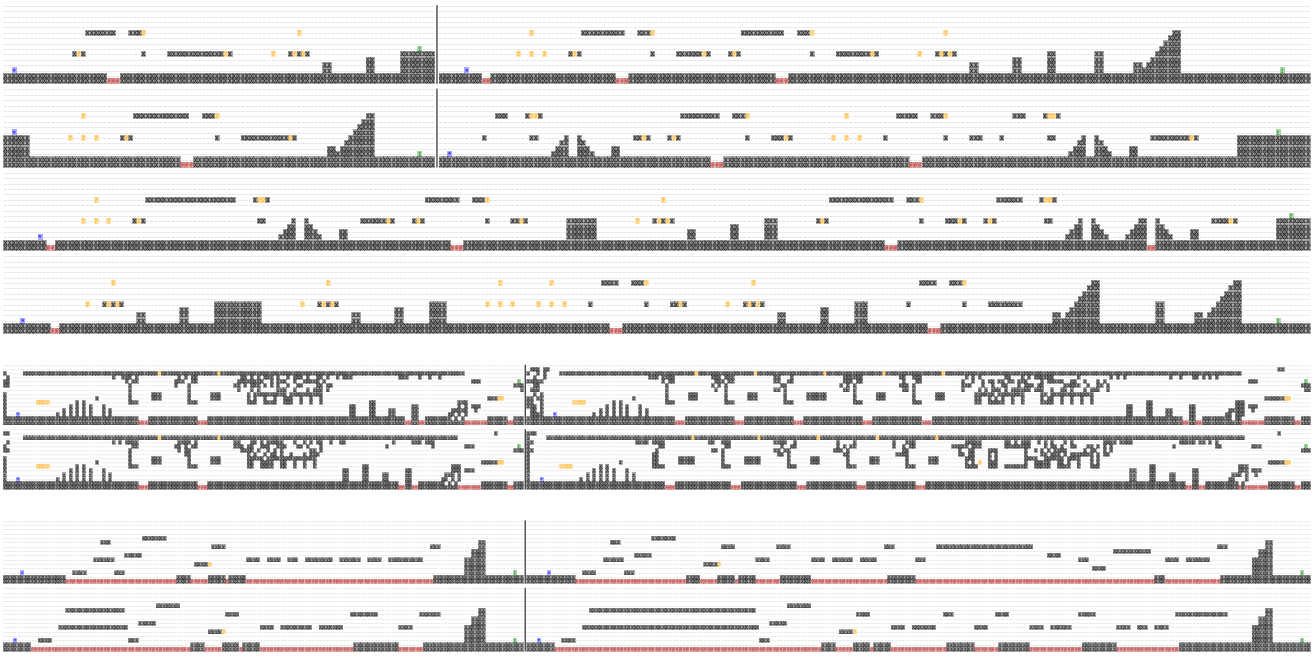

Figure 3: Example levels generated from the size test. From top to bottom, with two rows of each: 1-1, 50% and 100%; 1-1, 150%; 1-2, 100% and 150%; 1-3, 100% and 150%. Tile characters are `-`: blank, `X`: block, `@`: hazard, `?`: bonus, `*`: start, and `!`: goal. Border tiles omitted for clarity.

input level, scaled by the relative size of the output level (e.g. the counts are halved for levels generated at 50% size). This constraint helps prevent the solver from generating uninteresting levels like rectangles of blocks.

- *Platform reachability*: All blocks that do not have a block or a goal above them must have a reachable state in the tile above them with $isonground$ true. This prevents the solver from creating superfluous unreachable platforms in the air. We did not use this constraint when using SMB 1-2 for input, as that training level itself had many unreachable platforms.
- *Bonus reachability*: All bonuses must have a reachable state in the tile below, with $whichbonus$ set so they can be collected.

Processing took place on an AWS r5.4xlarge instance with 16 cores and 128GB RAM, using Python 3 and pypy 3. The clingo constraint solver was run with 12 threads, using a different random seed for each level generation.

To explore how different levels would impact the generation process, we used SMB levels 1-1, 1-2, and 1-3 for input. We ran two sets of level generation tests: a size test and a controllability test. To confirm playability, we (1) parsed the solver output to verify that a reachable path existed from the start to goal and (2) manually played all generated levels.

First, to explore the kinds of levels generated, how long it took, and how size impacted them, we generated levels of varying sizes: we used the input level height, and generated levels at 50%, 100% and 150% of the input level width. For each input level and size, we generated five levels. A summary of the size test and the levels generated is given in Table 1, and examples of levels generated in Figure 3.

Second, to explore how controllable the generated levels could be, we generated levels requiring exact counts of specific tile types. For each of the block, hazard, and bonus tile types, we tried generating a level that required an exact count for that tile type (while allowing the other two tile types to fall within the ranges previously discussed) for 100% size. A summary of the controllability test's results is given in Table 2, and examples of levels generated in Figure 4.

Discussion

Although we only requested five levels to be generated from each training level, we observed a few patterns across the generated levels. The generated levels were largely made up of repeated "motifs" that could be found in the training levels, such as stairs, pits, and groupings of platforms. These motifs are similar to the "scenes" with specific game mechanics that Green et al. (2020) showed could be stitched together to generate new SMB levels, though it should be noted that our approach ensures playability for all generated levels. The variety in our generated levels seems to be primarily based on reorganizations of these motifs with variations on their lengths. 1-2 had the least variety of levels generated, with generated levels of the same length being made up of the same sequences of motifs and minor rearrangements of blocks. In fact, 3 of the 5 levels generated from 1-2 at 100% size were identical. This may be partially attributable to the relatively closed-off nature of 1-2 and the lack of the platform reachability constraint, allowing the solver to create unusable platforms to easily satisfy the adjacency and tile type range constraints. The solver appeared more flexible in generating levels from 1-1 and 1-3,

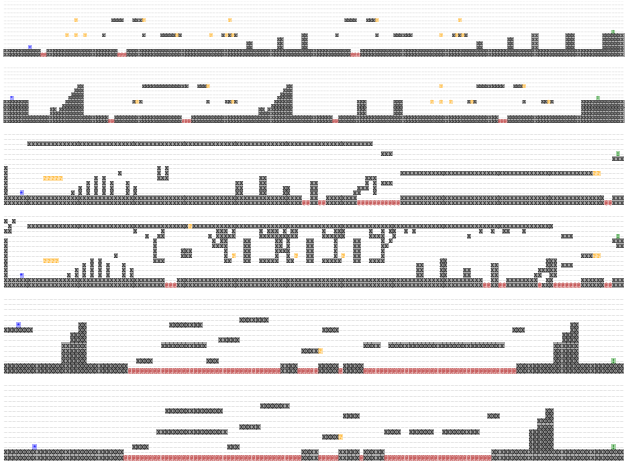

Figure 4: Example levels generated from the controllability test. From top to bottom: 1-1, 500 blocks; 1-1, 10 hazards; 1-2, 500 blocks; 1-2, 10 bonuses; 1-3, 300 blocks; 1-3, 75 hazards. Tile characters are `-`: blank, `X`: block, `@`: hazard, `?`: bonus, `*`: start, and `!`: goal. Border tiles omitted for clarity.

the training levels that had more empty space. 1-2 and 1-3 failed to generate smaller size levels, possibly due to not having enough room to work with to place metatiles.

We found that we needed to add some additional constraints to produce interesting and usable levels. Without the tile count constraints the solver could produce uninteresting levels such as simple rectangles of blocks. We also found that border tiles were needed to prevent the solver from finding undesirable solutions like levels with no bottom blocks to stand on (as it could exploit the fact that the neighbor metatile constraints only apply to metatiles that are present).

In terms of controllability, we found mixed results. Levels were only generated for about half of the tile count configurations we tested. This may be due to the limitations of the "motifs" discussed above: for example, in 1-1 there was no pit with only one hazard in it, and the solver could not generate a level with only one hazard. In practice, using range-based or soft constraints may be helpful to enable the solver to find valid solutions.

Limitations TLP has relatively simple physics-based movement rules. For example, the turtle accelerates when jumping or falling in the y-direction, but moves at a fixed constant velocity in the x-direction. We believe the general technique would work with a more complex movement physics simulation, but would result in a larger state graph to precompute. The turtle also has no animation state that might influence its movement, and the world itself is static.

The training and level generation process was memory and computation intensive, necessitating a powerful machine to run on. For example, the average grounding and solving process to generate a level from 1-1 at 150% width took nearly an hour. However, we have since been able to upgrade our approach to ground once and solve multiple times with different random seeds to generate multiple levels, thus reducing the time taken to generate a level.

Finally, although generated levels are technically playable, the path from start to goal can be difficult to follow (i.e. require precise timings for specific actions at exact locations). It may be interesting to be able to express the difficulty of following a path as a constraint as well.

Future Work Future work can explore games where the character has more complex movement rules, as well as more complex levels involving larger local neighborhoods (which would allow for the incorporation of moving elements like moving platforms and enemies) and generating levels by training on multiple levels at once. We could also consider other types of level generation primitives: in SMB-style levels, it may make sense to train on columns rather than individual tiles. Optimizations to improve the speed of training and level generation are also areas for future work.

Ethical Implications This research uses levels authored by humans, and future work based on it must grapple with ethical questions of compensation, privacy, and equity, among others. Due to space constraints, we focus here on compensation and equity, and refer readers to Metcalf and Crawford (2016) for a discussion of privacy concerns in ML.

In implementations and continuations of this research, people should be fairly compensated for their labor to make levels that train the system. Sloane et al. (2020) point out that many ML systems are built on uncompensated, unacknowledged labor. If misused, this could become one such system. If it produces profit or increases efficiency by supplanting work previously done by people, the profits or savings should be shared with the people who enabled them. Beyond monetary compensation, Sloane et al. (2020) call on implementers and researchers to ask whether their use of data empowers users or exploits them.

Moreover, machine learning may amplify harms in design (Phillips et al. 2016; Bennett and Keyes 2019): this system may create level elements that are harmful or inaccessible, or that reproduce intentionally abusive input. Often, machine learning focuses on removing "bias," but this is not sufficient: ML systems may reproduce hate symbols or add harmful elements, even if they are not "biased" towards them (Phillips et al. 2016). An equitable implementation would carefully vet input and output designs to minimize harm, especially to vulnerable or marginalized groups. If automated review is not capable of this detection, human review may be necessary. We further call for future work to undergo careful design and ethical review, going beyond this incomplete list of potential risks and engaging with intersectional analysis (Ciston 2019).

Conclusion

In this work, we present an approach for constraint-based platformer level generation that guarantees playability, based on player movement, without the need for post-generation evaluation and filtering. Our approach also supports additional constraints to offer designers more creative control and allow for flexible level generation. We believe this controllable, constraint-based PCGML technique can aid game designers in elaborating on new ideas, re-mixing and expanding on existing levels, and rapidly iterating, while maintaining playability.

References

Bauer, A. W., and Popović, Z. 2012. RRT-based game level analysis, visualization, and visual refinement. In *Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Bennett, C. L., and Keyes, O. 2019. What is the Point of Fairness? Disability, AI and The Complexity of Justice. In *ASSETS 2019 Workshop—AI Fairness for People with Disabilities*.

Ciston, S. 2019. Imagining Intersectional AI. In *7th Conference on Computation, Communication, Aesthetics & X*, 39.

Compton, K., and Mateas, M. 2006. Procedural Level Design for Platform Games. In *Second AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 109–111.

Dahlskog, S.; Togelius, J.; and Nelson, M. J. 2014. Linear levels through n-grams. In *Proceedings of the 18th International Academic MindTrek Conference: Media Business, Management, Content & Services*, 200–206.

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. ISBN: 0921-7126 Publisher: Citeseer.

Green, M. C.; Mugrai, L.; Khalifa, A.; and Togelius, J. 2020. Mario level generation from mechanics using scene stitching. *arXiv:2002.02992 [cs.AI]*.

Gumin, M. 2016. WaveFunctionCollapse. GitHub repository. https://github.com/mxgmn/WaveFunctionCollapse.

Guzdial, M., and Riedl, M. 2016. Game level generation from gameplay videos. In *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Guzdial, M.; Liao, N.; Chen, J.; Chen, S.-Y.; Shah, S.; Shah, V.; Reno, J.; Smith, G.; and Riedl, M. O. 2019. Friend, collaborator, student, manager: How design of an AI-driven game level editor affects creators. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 624:1–624:13.

Guzdial, M.; Liao, N.; and Riedl, M. 2018. Co-creative level design via machine learning. In *Experimental AI In Games Workshop*. arXiv:1809.09420.

Harrison, P. F. 2005. *Image Texture Tools*. Ph.D. Dissertation, Monash University.

Hoover, A. K.; Togelius, J.; and Yannakis, G. N. 2015. Composing video game levels with music metaphors through functional scaffolding. In *First Computational Creativity and Games Workshop*.

Horswill, I. D., and Foged, L. 2012. Fast procedural level population with playability constraints. In *Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Hoyt, A.; Guzdial, M.; Kumar, Y.; Smith, G.; and Riedl, M. O. 2019. Integrating Automated Play in Level Co-Creation. In *Experimental AI In Games Workshop*.

Karth, I., and Smith, A. M. 2017. WaveFunctionCollapse is constraint solving in the wild. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 68:1–68:10.

Karth, I., and Smith, A. M. 2018. Addressing the fundamental tension of PCGML with discriminative learning. *arXiv:1809.04432 [cs, stat]*.

Laskov, A. 2009. Level generation system for platform games based on a reinforcement learning approach. *University of Edinburgh, Tech. Rep. EDI-INF-IM090699*.

Londoño, S., and Missura, O. 2015. Graph Grammars for Super Mario Bros Levels. In *Proceedings of the 10th International Conference on the Foundations of Digital Games*.

Mawhorter, P., and Mateas, M. 2010. Procedural level generation using occupancy-regulated extension. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, 351–358.

Merrell, P. C. 2009. *Model synthesis*. Ph.D. Dissertation, University of North Carolina at Chapel Hill.

Metcalf, J., and Crawford, K. 2016. Where are human subjects in big data research? The emerging ethics divide. *Big Data & Society* 3(1).

Nelson, M. J., and Smith, A. M. 2016. ASP with applications to mazes and levels. In Shaker, N.; Togelius, J.; and Nelson, M. J., eds., *Procedural Content Generation in Games*, Computational Synthesis and Creative Systems. Cham: Springer International Publishing. 143–157.

Phillips, A.; Smith, G.; Cook, M.; and Short, T. 2016. Feminism and procedural content generation: toward a collaborative politics of computational creativity. *Digital Creativity* 27(1):82–97.

Sandhu, A.; Chen, Z.; and McCoy, J. 2019. Enhancing Wave Function Collapse with design-level constraints. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, 1–9.

Scurti, H., and Verbrugge, C. 2018. Generating paths with WFC. In *Fourteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Shaker, N.; Nicolau, M.; Yannakakis, G. N.; Togelius, J.; and O'Neill, M. 2012. Evolving levels for Super Mario Bros using grammatical evolution. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, 304–311.

Sloane, M.; Moss, E.; Awomolo, O.; and Forlano, L. 2020. Participation is not a design fix for machine learning. In *ICML 2020 Workshop on Participatory Approaches to Machine Learning*.

Smelik, R.; Tutenel, T.; de Kraker, K. J.; and Bidarra, R. 2010. Integrating procedural generation and manual editing of virtual worlds. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 1–8.

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.; Whitehead, J.; Mateas, M.; Treanor, M.; March, J.; and Cha, M. 2011. Launchpad: A rhythm-based level

generator for 2-d platformers. *IEEE Transactions on computational intelligence and AI in games* 3(1):1–16.

Smith, A. M.; Nelson, M. J.; and Mateas, M. 2010. Ludocore: A logical game engine for modeling videogames. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, 91–98.

Smith, G.; Whitehead, J.; and Mateas, M. 2010. Tanagra: A mixed-initiative level design tool. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, 209–216.

Snodgrass, S., and Ontañón, S. 2016. Controllable procedural content generation via constrained multi-dimensional Markov chain sampling. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, 780–786.

Snodgrass, S., and Ontañón, S. 2017. Learning to generate video game maps using Markov models. *IEEE Transactions on Computational Intelligence and AI in Games* 9(4):410–422.

Sorenson, N., and Pasquier, P. 2010. The evolution of fun: Automatic level design through challenge modeling. In *International Conference on Computational Creativity*, 258–267.

Stanton, M.; Humberston, B.; Kase, B.; O'Brien, J. F.; Fatahalian, K.; and Treuille, A. 2014. Self-refining games using player analytics. *ACM Transactions on Graphics (SIGGRAPH)* 33(4):73:1–73:9.

Stanton, M.; Geddert, S.; Blumer, A.; Hormis, P.; Nealen, A.; Cooper, S.; and Treuille, A. 2016. Large-scale finite state game engines. In *Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Computer Animation*.

Summerville, A., and Mateas, M. 2016. Super Mario as a string: platformer level generation via LSTMs. *arXiv:1603.00930 [cs]*.

Summerville, A. J.; Snodgrass, S.; Mateas, M.; and Ontañón, S. 2016. The VGLC: The Video Game Level Corpus. *arXiv:1606.07487 [cs]*.

Summerville, A.; Snodgrass, S.; Guzdial, M.; Holmgård, C.; Hoover, A. K.; Isaksen, A.; Nealen, A.; and Togelius, J. 2018. Procedural Content Generation via Machine Learning (PCGML). *IEEE Transactions on Games* 10(3):257–270.

Summerville, A. J.; Philip, S.; and Mateas, M. 2015. MCMCTS PCG 4 SMB: Monte Carlo tree search to guide platformer level generation. *Eleventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Togelius, J., and Dahlskog, S. 2013. Patterns as objectives for level generation. In *Proceedings of the Second Workshop on Design Patterns in Games*.

Yu, D., and Hull, A. 2009. Spelunky (PC Game).