Literally Unplayable: On Constraint-Based Generation of Uncompletable Levels

Seth Cooper
se.cooper@northeastern.edu
Northeastern University
Boston, USA

Mahsa Bazzaz
bazzaz.ma@northeastern.edu
Northeastern University
Boston, USA

ABSTRACT

Most research in procedural content generation has, understandably, focused on generating levels that are completable—that is, levels where it is possible for a player to complete them. In this work we explore the generation of uncompletable levels and their applications. Building on an existing constraint-based level generator, we add support for constraints that a level's goal is not reachable from its start. The generator can thus create levels that are similar to completeable levels in many ways (such as local tile patterns), yet are not possible to complete. We then describe several applications of those constraints and the resulting levels, including: qualitatively characterizing what makes levels uncompletable; creating training data for completability classifiers; checking that a generator can only generate completable levels; and generating levels that require the player to use a special move.

CCS CONCEPTS

• **Human-centered computing**;

KEYWORDS

procedural content generation, constraints, completability

ACM Reference Format:
Seth Cooper and Mahsa Bazzaz. 2024. Literally Unplayable: On Constraint-Based Generation of Uncompletable Levels. In *Proceedings of the 19th International Conference on the Foundations of Digital Games (FDG 2024), May 21–24, 2024, Worcester, MA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3649921.3659844

1 INTRODUCTION

When procedurally generating levels for games [26], the completability of the levels is important. Including a level that can't be completed in a game can be undesirable. Thus, most PCG research has understandably focused on generating completable levels. In contrast, in this work, we developed a level generator that only generates uncompletable levels, to explore possible applications of such a generator.

In this case, we use *completability* in particular to mean there is technically a way to solve the level, regardless of the difficulty of finding or carrying out the solution — although sometimes the term *playability* has been used. Additionally, we would like uncompletable levels to still have some reasonable structure and style to them and not just be something like random noise. Specifically, we look at 2D tile-based levels whose completablity is determined by there being a path from the start to the goal of the level, and generate levels using the Sturgeon constraint-based PCG system [5]. The system can learn the style of example levels by learning tile patterns from them. Thus we aim to generate levels that are locally similar to the examples, but which globally are not completable.

In this work we extended Sturgeon to incorporate additional constraints that can ensure that the goal is *not* reachable from the start, and use these in two games, a top-down exploration game and a side-view platformer. We give an overview of four potential uses of a level generator that can incorporate uncompletability constraints.

Qualitative Characterization of Uncompletable Levels: By generating a number of uncompletable levels, it may be possible to better understand some ways that a level can be uncompletable. This might help to improve generators of completable levels, or just give insight into levels or generator for a game.

Training Completability Classifiers: Machine learning models that can classify levels as completable or uncompletable may be useful in, for example, quickly filtering generated levels in place of a testing agent. Such classifiers would need training data including both completable and uncompletable levels.

Generating Completable Levels without Completability Constraints: If the constraints that a level be uncompletable are not satisfiable, then that means it is not possible for the generator to generate an uncompletable level. Thus we could remove those constraints (and any other completability constraints or checks) and have the generator generate completable levels regardless.

Generating Levels Only Completable with Special Moves: By generating levels that are completable with a special move, but uncompletable without it, we can generate levels that are only completable using the special move. This could be useful in games that introduce new moves to help teach them to players.

We provide an overview of these applications as an initial exploration of constraint-based generation of uncompletable levels. More work may be possible in each of them.

2 RELATED WORK

This work was partly inspired by recent graph layout work that used optimization for "the worst graph layout algorithm ever" [8]. We explored a similar concept in video games: a level generator that can only generate uncompletable levels. However, most prior

FDG 2024, May 21–24, 2024, Worcester, MA, USA

ACM ISBN 979-8-4007-0955-5/24/05
https://doi.org/10.1145/3649921.3659844

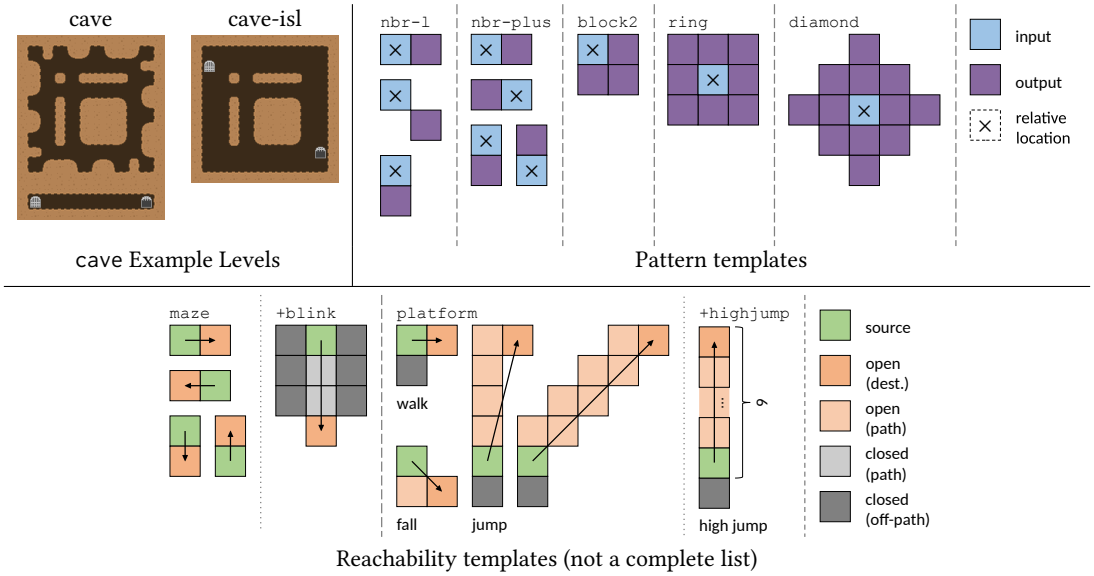

Figure 1: Some example levels, pattern templates, and reachability templates used.

Game	Setup Name	Size	Tile Counts	Example Level	Pattern Template	Reachability Template
mario (funct.)	`uncompletable`	14×32	at least 1 pipe, 1 gap, 34 block/ground, and 1-4 ? tiles	SMB 1-1	`ring`	`platform`:unreachable
	`completable`			SMB 1-1	`ring`	`platform`:reachable
	`must-with`			SMB 1-1	`diamond`	`platform`:reachable
	`must-without`			SMB 1-1	`diamond`	—
	`special`			SMB 1-1	`ring`	`platform`:unreachable & `platform+highjump`:reachable
(image)	—	—	—	SMB 1-1	`nbr-1`	—
cave	`uncompletable`	15×15	interior ~45–55% solid	cave	`nbr-plus`	`maze`:unreachable
	`completable`			cave	`nbr-plus`	`maze`:reachable
	`must-with`			cave-isl	`block2`	`maze`:reachable
	`must-without`			cave-isl	`block2`	—
	`special`			cave	`nbr-plus`	`maze`:unreachable & `maze+blink`:reachable

Table 1: Table of different setups used in this work.

work on procedural content generation for levels has focused on generating completable levels.

There are many constraint-based approaches to level generation [11, 15, 18, 24, 27, 30]. A number of these can incorporate constraints that levels are completable. For example, work in generating mazes or other levels using Answer Set Programming has incorporated constraints that there are paths through the level between desired areas [22]. The work of Smith et al. [28] used a related approach to generate solvable puzzles while ruling out undesirable solutions in an educational game. The Sturgeon system that our work builds on also includes path-based constraints [5], as well support for constraints that levels are completable using more complex rewrite-rule mechanics [6].

In search-based PCG, completablility can be incorporated into the fitness function, as generated levels can be tested for completablility, e.g. by agents [34]. One approach used in search-based PCG is feasible-infeasible two-population genetic algorithms (FI-2Pop) [17, 35, 37]. FI-2Pop evolves two populations, one feasible (e.g. completable) and the other unfeasible (e.g. uncompletable). While this approach does have the property that it can generate many uncompletable levels in the infeasible population, the intention is to generate completable ones. However, some work has presented unplayable levels found during search [36], and recent search-based PCG has used a fitness function to search for unplayable levels as a way to better understand the generator [9].

Machine learning approaches to PCG (i.e. PCGML [32]) have incorporated techniques to improve the completability of generated levels. For example, including path information in training levels

Start blocked (61): Only a few (usually 1) tiles are reachable near the start.

Start isolated (9): Only a few (usually 1) tiles are reachable near the start, which are in their own separate area of the level.

Middle blocked (2): A good amount of the level is reachable, with larger regions appearing reachable from start and goal.

Goal blocked (9): Most of the level is reachable, but goal and a few (usually 1) tiles near it are blocked.

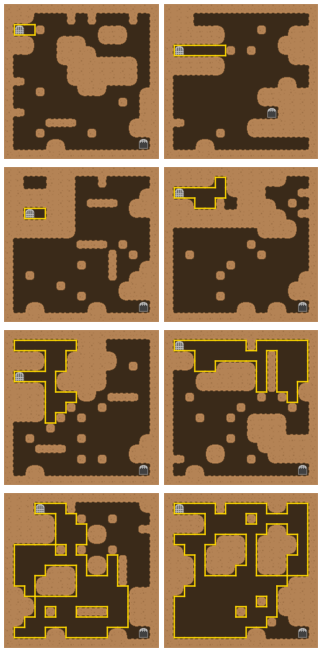

Figure 2: Uncompletable level categories for `cave`.

as a way to improve level quality and playability [25, 29, 31]. Some machine learning approaches to repairing uncompletable levels have been developed as well [14].

Other PCG work has taken different approaches to completability. Some have focused on taking pre-generated level segments or partial levels that are themselves completable, and assembling or extending those into larger completable levels [3, 20]. Some work goes further and ensures that not only are levels completable, but that it is not possible to get stuck [21].

3 SYSTEM OVERVIEW

In this work, we build on and use the Sturgeon constraint-based level generation system [5]. Sturgeon generates levels by converting higher-level "design rules" into constraint satisfaction problems, and using low-level solvers (such as SAT solvers) to find solutions. Two types of Sturgeon rules are most relevant to this work.

First, *pattern rules*, which learn local patterns of tiles from example levels, and constrain the generated levels to contain those patterns. Patterns are learned using *pattern templates*, which specify the relative relationships between tiles in learned patterns.

Second, *reachability rules*, which ensure that the goal of a level is reachable from the start. These rules are based on a *reachability template*, which specifies which tiles are potentially reachable from which other tiles. This creates a reachability graph over the level, with nodes representing locations and edges representing potential moves between locations. Edges require certain nodes (i.e. tiles) to be *open* (passable) or *closed* (solid) to be included in the solution path. Reachability rules add constraints that there is a path from the start to the goal in the reachability graph that meets the open and closed node requirements of the edges.

Pipe first (18): Path blocked by a pipe, which is the first obstacle.

Pipe further (35): Path blocked by a pipe, which is not the first obstacle.

Hill first (12): Path blocked by a hill, which is the first obstacle.

Hill further (5): Path blocked by a hill, which is not the first obstacle.

Gap first (1): Path blocked by a gap that cannot be jumped across, which is the first obstacle.

Gap further (6): Path blocked by a gap, which is not the first obstacle.

Gap and hill further (3): Path blocked by a gap with a hill on the far side, which are not the first obstacle.

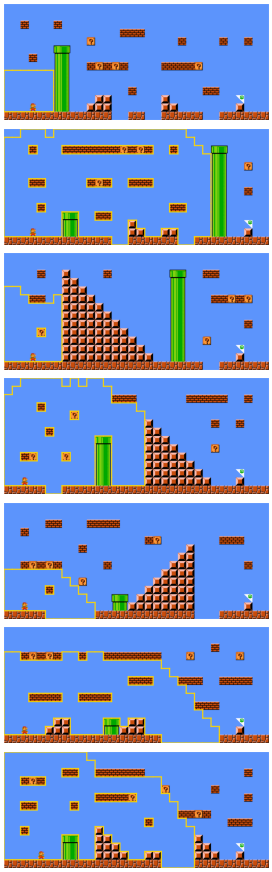

Figure 3: Uncompletable level categories for `mario`. (may have other blocks in the way that prevent the jump)

In this work specifically, we add a high-level *unreachability rule*, which uses the reachability graph to ensure that there is *not* a path from the start to the goal. This rule uses Sturgeon's mid-level API to set up the following constraints:

- If a node is the start node, it is reachable.
- If a node is the goal node, it is unreachable.
- If a node is closed, it is unreachable.
- If a node is open and has any reachable incoming edge, the node is reachable.
- If a node is reachable, any of its outgoing edges whose open/closed node requirements are met are reachable.

4 APPLICATIONS

To explore constraint-based generation of uncompletable levels, we applied unreachability rules to two games. Here we give a high-level view of the general setups used for each game (although some applications used more specific setups).

`cave` — A simple custom cave exploration game, using image tiles from Kenney [16]. Two custom example levels are used, and depending on the setup, the `nbr-plus` or `block2` pattern template is used. Tile count constraints are added to make the levels about half solid. The level start must be in the top-left and the goal in

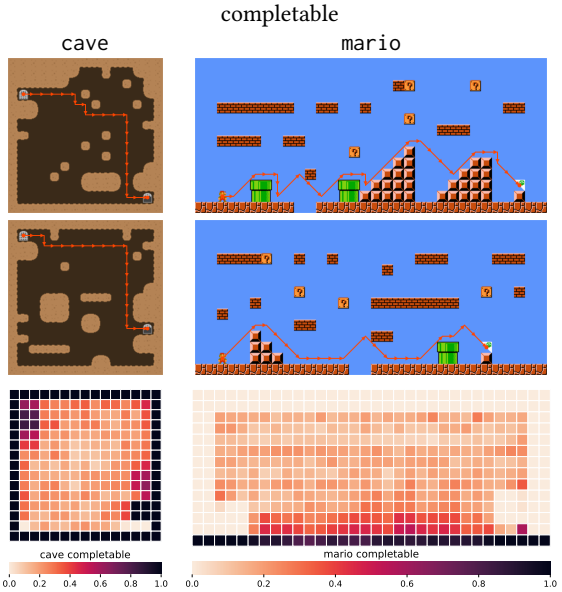

Figure 4: Sample `completable` levels and heatmap of solid tiles. A path through the level shown in red.

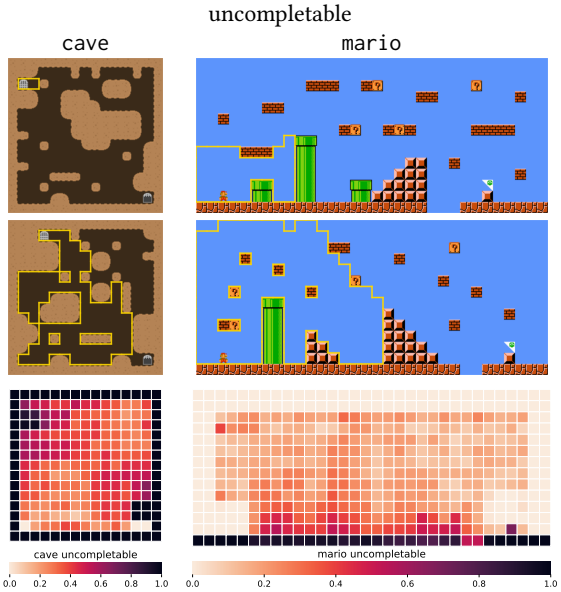

Figure 5: Sample `uncompletable` levels and heatmap of solid tiles. Boundary of what is reachable shown in yellow.

the bottom-right. The main reachability template used is the `maze` reachability template, which represents top-down movement.

`mario` — Based on the game *Super Mario Bros.* A cleaned-up version of level 1-1 from the VGLC [33] is used as the example level. Tile count constraints are added to make sure there are several interesting elements in the level. Levels are generated in a two-step process: first, the functional part of the level is generated using the larger `ring` or `diamond` pattern templates, and then the level image is generated, using the `nbr-1` pattern templates. The level start must be on the left and the goal on the right. The main reachability template used is the `platform` template, which represents walking, falling, and jumping.

A summary of all setups is given in Table 1; although several different setups were used depending on the application, some further information on specific setups is given when the application that used them is discussed. Example levels and templates used are shown in Figure 1. Timing information is given in Figure 6, discussed in the applications. 80 levels were generated for each setup. The levels are available at https://osf.io/td4nb/.

To solve for levels, we used a portfolio solver consisting of three instances of PySAT's [12] MiniCard [19] solver. In screenshots of the levels, when there is no path from the start to the goal, the boundary of what is reachable is shown in yellow, and when there is a path, it is shown in red. Since the solution paths output by Sturgeon can be very indirect, we re-ran a pathfinder to find more direct paths from the start to the goal in completable levels, or find reachable locations to confirm the goal is unreachable in uncompletable levels.

Next, we describe several different applications of using unreachability rules to generate uncompletable levels. We describe the approach and discuss the results in each application individually.

4.1 Qualitative Characterization of Uncompletable Levels

Approach — One application of uncompletable levels is to qualitatively characterize what makes levels uncompletable. For this application, we generated levels using the `uncompletable` setup, which incorporates the unreachability rule constraints to ensure there is no path from the start to the goal. Sample levels are shown in Figure 5. We categorized the levels based on what made them uncompletable, with a focus on the boundary of the reachable area and what was preventing further areas from being reachable. This was done informally by the authors first independently, then based on discussion.

In `cave`, the categories were largely based on how far the player could make it into the level before being blocked, either right at the beginning, right before the end, or somewhere in the middle. Some also had the start appearing isolated in its own separate area of the level. The large majority of levels had the start blocked within a few tiles, usually only one.

In `mario`, the categories were based on what type of obstacle blocked progress, either a pipe or hill that was too tall to jump over, or a gap that could not be jumped across. Sometimes other blocks were in the way of a jump. A few levels incorporated a gap and a hill together. These categories were also divided into if they were the first obstacle the player encountered or they came later in the level. Pipes were the most common obstacle to block progress, possibly due to their flexible placement and size.

Discussion — In this application, we were able to determine several categories of what makes levels uncompletable. In both games we found it interesting to note how far it appeared the player could make it through the level before being blocked. Categorization was done manually, but could possibly be done in an automated way

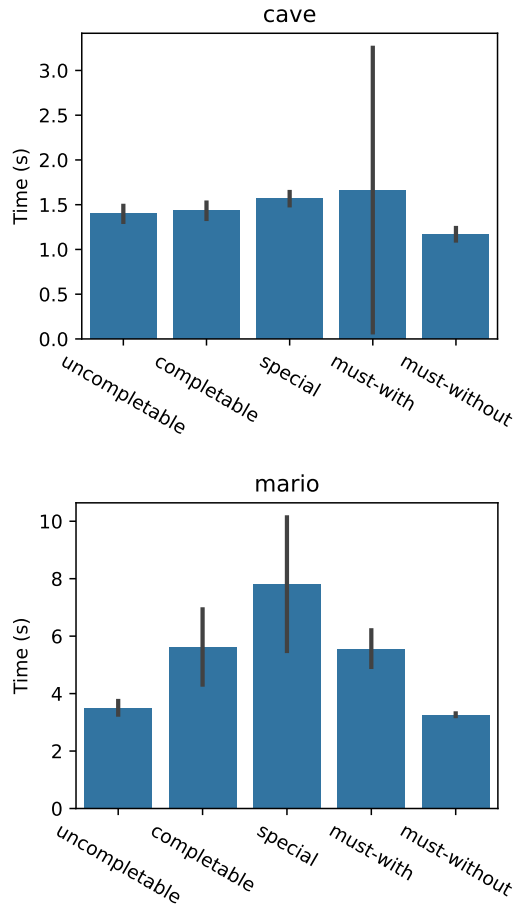

Figure 6: Times for generating levels with different setups. Mean shown with standard deviation error bars.

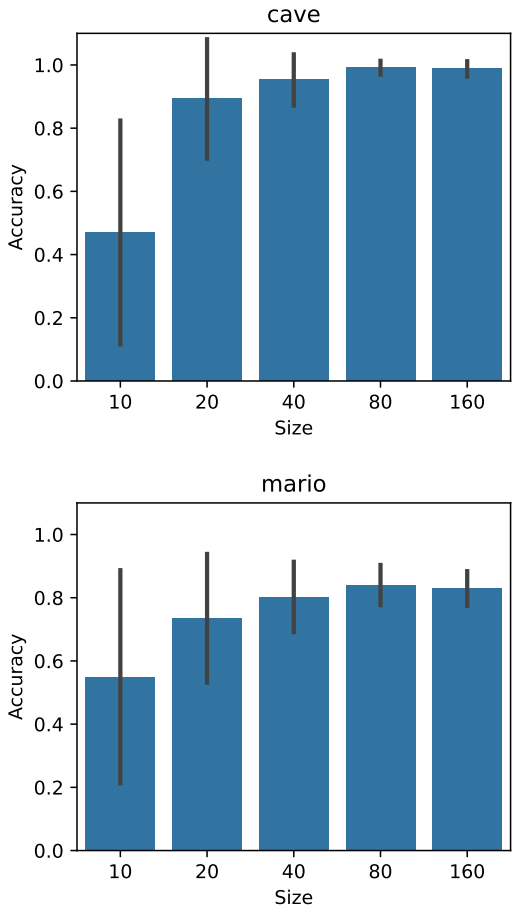

Figure 7: Accuracy of classifier of `completable` vs. `uncompletable` levels for different dataset sizes. Mean shown with standard deviation error bars.

in the future. If incorporated with machine learning approaches, it might be possible to incorporate into explainability techniques.

The categories we determined were subjective, and there are other categorizations possible; even with these categories there can be some subjectivity about resolving which category a specific level is in for ambiguous cases. These are also based just on the setup specific and generator used, and it's possible, for example, other pattern templates or even other solvers might have different categories or distributions of levels across categories. These are also not exhaustive, as there may be rare levels that did not show up in our sample, but would if we generated more levels.

4.2 Training Completability Classifiers

Approach — Another application of uncompletable levels is as training data for machine learning completability classifiers. Such a classifier will need samples of both completable and uncompletable levels for training, and being able to directly generate uncompletable levels could help with this process. Previous work, for example, has used active learning to train such a classifier [2], but with a pool-based approach.

Thus, for this application we also used level generated with the `completable` setup, which uses reachability rules to ensure a path from the start to the goal. Sample levels are shown in Figure 4. We then used increasing numbers of levels from the `completable` and `uncompletable` levels to examine the impact of dataset size on classifier accuracy.

Our goal was mainly to examine the impact of dataset sizes, and not necessarily to develop the best possible classifier. For each dataset size, we selected an equal number of `completable` and `uncompletable` levels for a balanced dataset. We used random forest binary classifiers with 100 trees based on the example provided in the VGLC [33] implemented with scikit-learn [23]. The features for the classifier were the frequencies of occurrences of all 2×2, 3×3, 4×4 square patterns in the training levels. With 5-fold cross-validation, we trained 20 random forest classifiers on the training data and determined accuracy on the test data, resulting in accuracies for 100 folds for each dataset size. Given the balanced dataset sizes, accuracy of 0.5 would be a reasonable baseline. Results are shown in Figure 7.

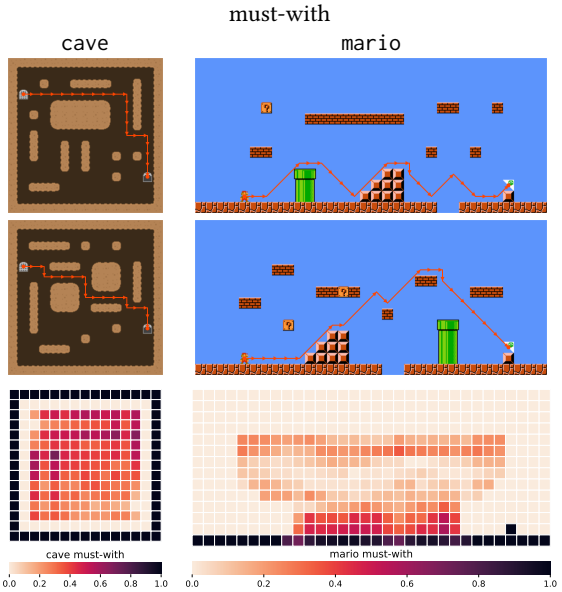

Figure 8: Sample `must-with` levels and heatmap of solid tiles. A path through the level shown in red.

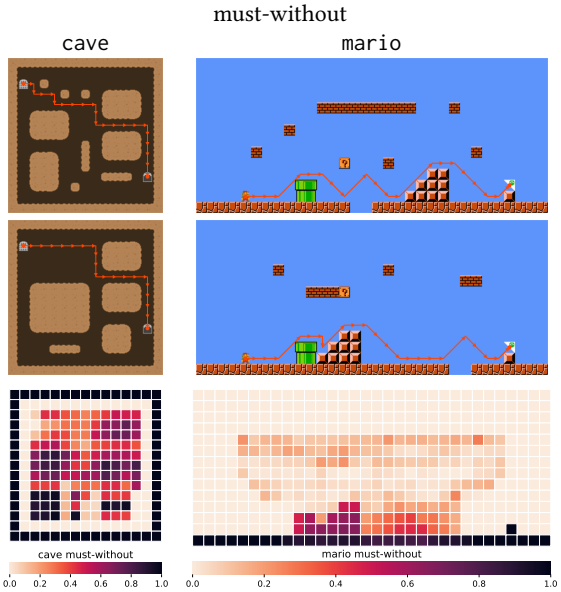

Figure 9: Sample `must-without` levels and heatmap of solid tiles. A path through the level shown in red.

Discussion — The classifier accuracy does appear to go up as the dataset size increases. Both games start near 0.5 with only a few examples, but quickly increase. Even with a dataset of only 160 levels, the `cave` classifer does very well, surpassing an average of 0.95 accuracy, while the `mario` classifier passes an average of 0.8 accuracy.

With samples from two setups, the time to generate `completable` and `uncompletable` levels can be compared in Figure 6. For `cave` they appear comparable, but for `mario` uncompletable levels appear to generate more quickly. Similarly, heatmaps of solid tiles can be compared in Figures 5 and 4. Though similar, it does appear that the uncompletable levels use more solid tiles, particularly high up with tall obstacles in `mario`. However, `uncompletable` `mario` levels also appear to have fewer solid tiles along the bottom near the end, likely where gaps are introduced.

Again, this analysis is based only on this level generator, and it is possible that other generators might make levels that are harder to classify. The classification features used were similar to what is used to generate the levels — local tile patterns. Also, `mario` accuracy does appear to level off, and a more sophisticated classifier might be needed to improve accuracy further.

4.3 Generating Completable Levels without Completability Constraints

Approach — We can also use the unreachability rules to check if all the levels generated will be completable. That is, if the unreachability rules are used, and a level cannot be generated, then this means that the unreachability rule constraint cannot be satisfied (presuming that a level can be generated at all without the unreachability rules). If this is the case, any generated level must have the goal reachable from the start. Logically, we might say: if there does not exist a level that is uncompletable, then all levels must be completable. Thus, levels can be generated *without* the reachability rules and still must be completable. This may improve performance as the number of the constraints to be solved can be greatly reduced, as the solver does not also need to solve for a path through the level.

To explore this application, we found setups for each game that could generate levels, but could not generate levels when the unreachability rules were applied. Finding these setups was guided by the authors' experience with the Sturgeon system. For `cave`, this involved a new example level that had only "islands" of solid tiles among open space (shown in Figure 1), and used the `block2` pattern template. For `mario`, this was simply using the larger `diamond` pattern template to capture more context around each tile in the example level.

Using these setups, we then generated levels both with reachability rules (`must-with`) and without reachability rules (`must-without`). Sample levels and solid tile heatmaps are shown in Figures 8 and 9.

Discussion — This approach appears to work to determine if a setup can only generate completable levels. From the timing information in Figure 6, it appears the `must-without` setups do generate levels faster than the `must-with`. Looking at the heatmaps in Figures 8 and 9, these levels, unsurprisingly, have more open space in them than other levels. Comparing just these two setups, the distribution of solid tiles appears shifted, with some areas more often containing solid tiles in `must-without` levels. This may indicate less overall variety in these levels. So while the levels may generate faster, there is certainly an impact on the structure of the levels themselves, which may or may not be desirable. Thus, a designer may not necessarily want to adjust their setup to find one such as we did. However, if a given setup is already what a designer

wants, using unreachability rules to check that it only generates completable levels is straightforward.

4.4 Generating Levels Only Completable with Special Moves

Approach — It is possible to use unreachability rules simultaneously with reachability rules to generate levels that are completable with one moveset but not another. This could be used to generate levels that are only completable using a special move; for example, in cases where a designer wants to require the player to use a special move, possibly for tutorial purposes. Both unreachability rules for a template without the special move *and* reachability rules for a template with the special move can be applied. Then, levels where the goal is unreachable without the special move, but reachable with it, will be generated.

Some previous work has, for example, worked toward automatically finding special moves or mechanics that can complete given levels [4, 10], whereas in this application we aim to generate levels based on players with different movesets. Thus this approach may be considered to have some commonality with restricted play [13], which has previously been used for game balancing, where the moveset without the special move is restricted.

For this application, we created a reachability template for each game that had all the moves of the original, but also added a special move. For `cave`, this was a "blink" move that allowed the player to teleport through exactly two solid tiles when also surrounded by solid tiles. In `mario`, the special move was a highjump that was much higher than the original jumps. These additions to the reachability templates are shown in Figure 1. We then generated levels using these `special` setups. Samples shown in Figure 10.

Discussion — This approach did allow generating levels only completable using a special move. In `mario`, it was not uncommon for the path to use a highjump to get onto a block to then reach higher points in the level. Looking at timing in Figure 6, these levels took notably longer to generate, likely due to the additional constraints to be solved. The solid tile heatmaps appear to use more solid tiles, similar to the `uncompletable` setup.

4.5 Other applications

While we have explored a few applications here, we think there could be further applications to intentionally generating uncompletable levels. These could be used to test or make benchmark data sets for level repair algorithms (e.g. [7, 38]). Uncompletable levels could also be used as training examples for generative models that can incorporate positive and negative examples (e.g. [1]).

5 CONCLUSION

In this work we presented the intentional generation of uncompletable levels and several related applications. The levels, though uncompletable, still capture the structure and style of example levels by learning tile patterns. Uncompletable levels were generated by using a constraint-based generation approach that incorporates additional constraints that a level's goal is not reachable from its start. These unreachable rules allowed applications in qualitatively characterizing what makes levels uncompletable; creating training data for completability classifiers; checking that a generator can only generate completable levels; and generating levels that require the player to use a special move.

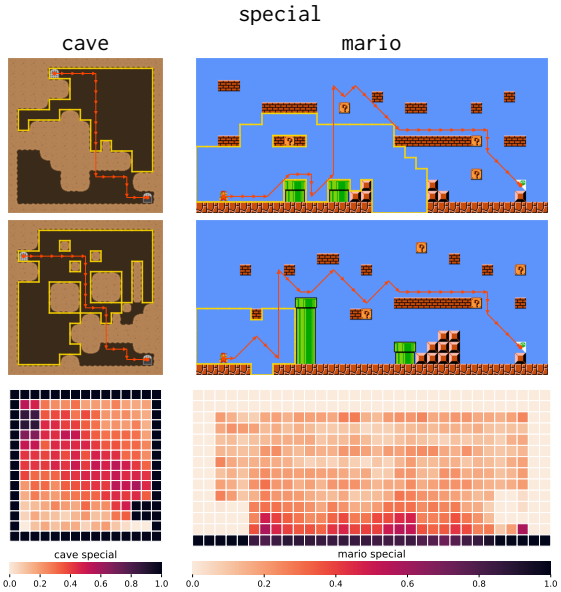

Figure 10: Sample `special` levels and heatmap of solid tiles. Boundard of what is reachable by the base player shown in yellow, and a path through the level by the player with a special move shown in red.

In the future, we are interested in further exploring these applications, such as more expressive range analysis, or the behavior of different solvers other than the PySAT solver used here.

It may also be interesting to explore what makes other types of generated content — such as music, quests, or narrative — the "worst" or unusable in some way, while still being stylistically coherent, to better understand the boundary between usable and unusable content.

REFERENCES

[1] Siddarth Asokan and Chandra Seelamantula. 2020. Teaching a GAN what not to learn. In *Advances in Neural Information Processing Systems*, Vol. 33. Curran Associates, Inc., 3964–3975.
[2] Mahsa Bazzaz and Seth Cooper. 2023. Active Learning for Classifying 2D Grid-Based Level Completability. In *2023 IEEE Conference on Games (CoG)*.
[3] Colan F. Biemer and Seth Cooper. 2022. On linking level segments. In *2022 IEEE Conference on Games (CoG)*. 199–205.
[4] Michael Cook, Simon Colton, Azalea Raad, and Jeremy Gow. 2013. Mechanic Miner: reflection-driven game mechanic discovery and level design. In *Applications of Evolutionary Computation*. 284–293.
[5] Seth Cooper. 2022. Sturgeon: tile-based procedural level generation via learned and designed constraints. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 18, 1 (2022), 26–36.
[6] Seth Cooper. 2023. Sturgeon-MKIII: simultaneous level and example playthrough generation via constraint satisfaction with tile rewrite rules. In *Proceedings of the 18th International Conference on the Foundations of Digital Games*.
[7] Seth Cooper and Anurag Sarkar. 2020. Pathfinding Agents for Platformer Level Repair. In *Proceedings of the Experimental AI in Games Workshop*.
[8] Sara Di Bartolomeo, Matěj Lang, and Cody Dunne. 2022. The worst graph layout algorithm ever. In *Proc. alt.VIS workshop at IEEE VIS (alt.VIS)*.
[9] Maria Edwards, Ming Jiang, and Julian Togelius. 2021. Search-based exploration and diagnosis of TOAD-GAN. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Vol. 17. 140–147.

[10] Johor Jara Gonzalez, Seth Cooper, and Matthew Guzdial. 2023. Mechanic Maker 2.0: reinforcement learning for evaluating generated rules. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 19, 1 (Oct. 2023), 266–275.

[11] Maxim Gumin. 2016. WaveFunctionCollapse. https://github.com/mxgmn/WaveFunctionCollapse.

[12] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. 2018. PySAT: a Python toolkit for prototyping with SAT oracles. In *Theory and Applications of Satisfiability Testing – SAT 2018*. 428–437.

[13] Alexander Jaffe, Alex Miller, Erik Andersen, Yun-En Liu, Anna Karlin, and Zoran Popovic. 2012. Evaluating competitive game balance with restricted play. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 8, 1 (Oct. 2012).

[14] Rishabh Jain, Aaron Isaksen, Christoffer Holmgård, and Julian Togelius. 2016. Autoencoders for level generation, repair, and recognition. In *Proceedings of the ICCC workshop on computational creativity and games*, Vol. 9.

[15] Isaac Karth and Adam M. Smith. 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.

[16] Kenney. 2022. Free game assets. https://www.kenney.nl/assets.

[17] Ahmed Khalifa, Michael Cerny Green, Gabriella Barros, and Julian Togelius. 2019. Intentional computational level design. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 796–803.

[18] Vivian Lee, Nathan Partlan, and Seth Cooper. 2020. Precomputing player movement in platformers for level generation with reachability constraints. In *Proceedings of the Experimental AI in Games Workshop*. 8.

[19] Mark H. Liffiton and Jordyn C. Maglalang. 2012. A cardinality solver: more expressive constraints for free. In *Theory and Applications of Satisfiability Testing – SAT 2012*. 485–486.

[20] Hao Mao and Seth Cooper. 2022. Segment-wise level generation using iterative constrained extension. In *2023 IEEE Conference on Games (CoG)*.

[21] Ross Mawhorter and Adam Smith. 2021. Softlock detection for Super Metroid with computation tree logic. In *The 16th International Conference on the Foundations of Digital Games (FDG) 2021*. 1–10.

[22] Mark J. Nelson and Adam M. Smith. 2016. ASP with applications to mazes and levels. In *Procedural Content Generation in Games*, Noor Shaker, Julian Togelius, and Mark J. Nelson (Eds.). Springer International Publishing, 143–157.

[23] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: machine learning in Python. *Journal of Machine Learning Research* 12, 85 (2011), 2825–2830.

[24] Arunpreet Sandhu, Zeyuan Chen, and Joshua McCoy. 2019. Enhancing Wave Function Collapse with design-level constraints. In *Proceedings of the 14th International Conference on the Foundations of Digital Games (FDG '19)*. Association for Computing Machinery, San Luis Obispo, California, 1–9.

[25] Anurag Sarkar, Adam Summerville, Sam Snodgrass, Gerard Bentley, and Joseph Osborn. 2020. Exploring level blending across platformers via paths and affordances. In *Proceedings of the Sixteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'20)*. AAAI Press, 280–286.

[26] Noor Shaker, Julian Togelius, and Mark J. Nelson. 2016. *Procedural Content Generation in Games*. Springer International Publishing.

[27] Adam M. Smith, Erik Andersen, Michael Mateas, and Zoran Popović. 2012. A case study of expressively constrainable level design automation tools for a puzzle game. In *Proceedings of the International Conference on the Foundations of Digital Games (FDG '12)*. 156–163.

[28] Adam M. Smith, Eric Butler, and Zoran Popovic. 2013. Quantifying over Play: Constraining Undesirable Solutions in Puzzle Design. In *Proceedings of the 8th International Conference on Foundations of Digital Games*. 221–228.

[29] Sam Snodgrass and Santiago Ontañón. 2017. Procedural level generation using multi-layer level representations with MdMCs. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. 280–287.

[30] Sam Snodgrass and Santiago Ontañón. 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.

[31] Adam Summerville and Michael Mateas. 2016. Super Mario as a string: platformer level generation via LSTMs. *arXiv:1603.00930 [cs]* (March 2016). arXiv:1603.00930 [cs]

[32] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K. Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. 2018. Procedural Content Generation via Machine Learning (PCGML). *IEEE Transactions on Games* 10, 3 (Sept. 2018), 257–270.

[33] Adam James Summerville, Sam Snodgrass, Michael Mateas, and Santiago Ontañón. 2016. The VGLC: The Video Game Level Corpus. *arXiv:1606.07487 [cs]* (July 2016). http://arxiv.org/abs/1606.07487

[34] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. 2011. Search-based procedural content generation: a taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 172–186.

[35] Breno M. F. Viana, Leonardo T. Pereira, Claudio F. M. Toledo, Selan R. dos Santos, and Silvia M. D. M. Maia. 2022. Feasible–infeasible two-population genetic algorithm to evolve dungeon levels with dependencies in barrier mechanics. *Applied Soft Computing* 119 (April 2022), 108586.

[36] Vanessa Volz, Jacob Schrum, Jialin Liu, Simon M Lucas, Adam Smith, and Sebastian Risi. 2018. Evolving Mario levels in the latent space of a deep convolutional generative adversarial network. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 221–228.

[37] Adeel Zafar, Hasan Mujtaba, and Mirza Omer Beg. 2020. Search-based procedural content generation for GVG-LG. *Applied Soft Computing* 86 (Jan. 2020), 105909.

[38] Hejia Zhang, Matthew Fontaine, Amy Hoover, Julian Togelius, Bistra Dilkina, and Stefanos Nikolaidis. 2020. Video game level repair via mixed integer linear programming. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 16, 1 (Oct. 2020), 151–158.