Stuck in the Middle: Generating Levels without (or with) Softlocks

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

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

Abstract

When generating game levels, it is desirable for them to be completable. Depending on the designer's goals, it may also be desirable to generate levels without softlocks. A softlock is a situation where the player has not won or lost, but cannot make progress toward the goal, and is thus stuck (unless they reset, or possibly lose the level). In this work we present a constraint-based approach to generating 2D tile-based levels that are completable and do not have areas where the player can get stuck. The approach uses a constraint-based reachability categorization of locations during level generation. This categorization can be used to prevent softlocks by ensuring that all locations reachable going forward from the start of the level are also reachable going backward from the goal, unless they are sinks (areas where the player will inevitably lose the level, e.g. falling off the bottom). Using this approach, it is also possible to intentionally generate levels with softlocks and to repair levels to remove softlocks.

Keywords

video games, procedural content generation, constraints, softlocks

ACM Reference Format:
Seth Cooper and Mahsa Bazzaz. 2025. Stuck in the Middle: Generating Levels without (or with) Softlocks. In *International Conference on the Foundations of Digital Games (FDG '25), April 15–18, 2025, Graz, Austria.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3723498.3723844

1 Introduction

A popular application of procedural content generation [34] is the automated generation of video game levels. When generating levels automatically, it is generally desirable that they be completable — that is, it should be technically possible for a player to complete the level. In a game where the player navigates through the level, for example, this would mean that there is some path from the start to the goal. There are many approaches to ensuring completability, such as having an agent test [43] or repair [46] a level after it is generated, or constraining a level to be completable [7].

Although checking (or constraining) that a solution exists will confirm that a level is completable, it may still be possible for the player to get stuck in some way if they stray from the solution path. This situation is often referred to as a *softlock*: the player has not won or lost, but it is no longer possible to complete the level. That is, even if the level can be completed when starting from the beginning, it may not be possible to complete from all the places it is possible to get to. Game players have long been interested in softlocks, and if the games they are playing have them. For example, the "Cruelty Scale" gives games a rating from "Merciful" to "Cruel" based on if it is possible to get stuck, and how obvious it is that this will happen (or has already happened) [32].

In this work, we developed a constraint-based approach to generating 2D tile-based levels where the player cannot get softlocked while navigating. It was developed using the Sturgeon constraint-based level generation system [7]. At a high level, we integrate constraints that categorize locations in the level as *forward reachable* (reachable by going forward from the *start* — i.e. all those locations that the player can get to), *backward reachable* (reachable by going backward from the *goal* — i.e. all those locations from which it is possible to complete the level), and *sink* (locations at which the player will inevitably lose the level by reaching a *hazard*). The constraints are based on a Sturgeon's *reachability graph* through the level that specifies how the player can move through the level for a given game.

With this location categorization, additional specialized constraints can be added to produce levels with desired properties. For example, to generate a level where the player cannot get stuck in a softlock, we constrain that all forward reachable locations that are not sinks are backward reachable — that is, if a player can get somewhere in the level, they can either still get to the goal from there, or they will inevitably lose the level.

While levels without softlocks may be generally desirable, designers may have other objectives for their levels. For example, in "masocore" games designed to challenge players with high difficulty [22], softlocks, or even impossible levels, may be desirable. Thus we also explored generating levels with softlocks and impossible levels using our approach.

Since our approach is using reachability, we are only considering softlocks based on player movement, and not other possible game mechanics such as, e.g. powerups. Also, situations where the player cannot move at all, but has not yet lost, we consider to be softlocks, although sometimes this may be distinguished as a separate kind of lock (e.g. deadlock).

We applied the approach to three different games with different types of movement: `driller`, a game where the player digs down through dirt; `slide`, a game where the player slides on ice; and `mario`, a platformer. We compared to the prior-used path-based completability. We found the new reachability approach is flexible enough for a variety or applications, although we found it to take up to 5 times longer on average to generate levels without softlocks, and even slower for specific applications. We also found that in most cases, it appears to increase the range of levels generated.

This work is licensed under a Creative Commons Attribution International 4.0 License.

FDG '25, Graz, Austria
© 2025 Copyright held by the owner/author(s).
ACM ISBN /25/04
https://doi.org/10.1145/3723498.3723844

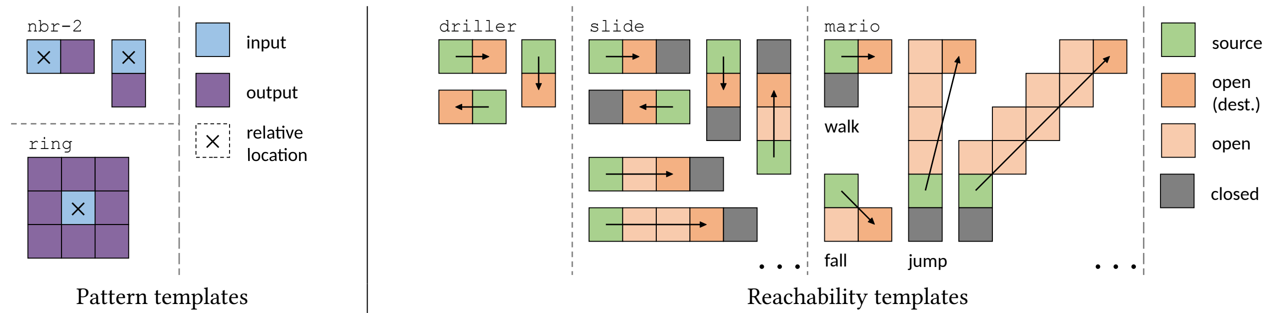

Figure 1: Pattern and reachability templates used in this work. Pattern templates are used to learn tile patterns from example levels. Each input tile constraints the associated output tiles to match those seen in the example level. Reachability templates determine edges in the reachability graph of the level. An edge is considered traversable if the desired configuration of open and closed tiles is met. Not all parts of reachability templates are shown.

The contributions of the this work include: a constraint-based formulation of categorizing types of reachability in a 2D tile-based levels; a demonstration of using these categorizations for various effects, including preventing softlocks; and an application of these to different games.

2 Related Work

2.1 Procedural Content Generation

There has a been a wide breadth of research in procedural content generation [34]. Most closely related to our current work are constraint-based approaches to generating levels. Since our approach can learn tile patterns from example levels, it is also related to procedural content generation via machine learning techniques [39]. Research in softlocks and games is discussed more in the subsequent subsection.

When generating levels using constraint solving, Answer Set Programming has proven popular [23, 30, 35]. Additionally, search- [13], sampling- [37], and SAT-based [7] approaches are used. Some work has looked at using constraints to prevent undesirable solutions in generated levels [36]. Recently, the popularity of the WaveFunctionCollapse algorithm [17] has motivated techniques that learn tile neighborhood constraints from example levels [7, 23, 24].

Completabilty of levels has also received much attention in level generation research. Rather than ensuring it is not possible to get stuck entirely, this work usually considers just ensuring a single path through the level exists. A common approach is using generate-and-test with an agent, potentially incorporating completability by the agent into a search's objective function [37, 43]. Some constraint-based approaches have been taken [7, 30] as well as using a generate-and-repair approach [10, 46].

2.2 Detecting Locks

In the domain of testing and analyzing software systems (often concurrent software), there has been much work on the use of both forward and backward reachability [19]. This includes checking for situations where tasks can become permanently blocked [42]. In these approaches, it is often desired to determine if a system will reach some undesirable (or unsafe) state. Generally, the approach is to check if any undesirable states are forward reachable from starting states, or, if starting states are backward reachable from any undesirable states [29, 38]. Some work in this are has used backward reachability to determine program states with the "liveness" property [1], where liveness is generally considered that 'something good' eventually happens [3]. In addition, some work has looked at comparing or combining forward and backward reachability [29, 31, 38, 44]. Since in our work we are using a constraint solver to check properties of a finite number of steps of a system, it may be considered related to bounded model checking [5].

Work in planning also considers generating plans using forward and/or backward approaches, roughly described as either searching forward from a starting state or searching backward from a goal state [27]. Some planning work has looked at combining both forms of search [16].

Our work might be considered to be using backward analysis from the goal (and sink analysis) to identify the undesirable states (i.e. those that are neither backward reachable nor sink), combined with forward analysis to make sure these states are not reachable from the start.

As players getting into a situation where they are can still play but not complete the level (i.e. a softlock) is usually considered undesirable and a bug in the game [11],there has been some work specifically in games in detecting softlocks (or relatedly, deadlocks). Some approaches use models of the games to look for bugs, including Petri nets [2, 33], hyperstate space graphs [6], and computation tree logic [28]. Similarly, there has been recent work in automated playtesting [14, 18], using agents to check for potentially undesirable states in levels, including specifically looking for places where the player can get stuck [4, 15, 45].

These game-related approaches are usually applied to levels that have already been created, often by a human designer, whereas our approach integrates the prevention of softlocks directly into the level generator. Some recent work has, though, taken softlocks into consideration when constructively generating levels [26].

3 Methodology

3.1 Sturgeon

This work is based on Sturgeon [7], a system for constraint-based generation of 2D tile-based levels. Here we describe the system's

features most relevant to the current work. At a high level, Sturgeon takes in a variety of design constraints on the level to be generated, translates them down to a constraint satisfaction problem (generally using Boolean variables), uses a constraint solver (e.g. SAT solver) to solve the problem, and then uses the solution to make the generated level. Using this approach, many constraints can be added on top of the base level generation constraints.

Sturgeon can learn to generate levels with local arrangements of tiles similar to an example level using *pattern templates*. A pattern template describes the relationship between *input* and *output* tiles that must be maintained: for an input tile at a particular location in a generated level, the related output tiles must have been seen in the example level. Given a pattern template and an example level, relevant arrangements of tiles are extracted and used to constrain the generated level. The pattern templates used in this work are shown in Figure 1. The `nbr-2` pattern template requires that the tiles on the left and bottom must have been in the example level; the `ring` pattern template requires that all eight neighboring tiles must have jointly been in the example level.

To represent movement for a game, Sturgeon uses a *reachability template*. For the purposes of basic reachability, tiles are considered *open* (the player can move through them) or *closed* (the player cannot move through them). A reachability template describes how the player can move through the level, given a relative local arrangement of open and closed tiles. For a specific level to be generated, the template is used to construct a *reachability graph*: a directed graph, where nodes are locations, and edges represent potential moves the player might make. An edge whose arrangement of open and closed nodes (i.e. locations) is matched is considered *traversable* and could be part of a valid path through the level. The reachability templates used in this work are shown in Figure 1; they are described below with the games they relate to.

3.2 Reachability Categorization

The primary addition to Sturgeon in this work is a constraint-based categorization of the reachability of locations in a level, as it is being generated. This constraint problem is included in the same overall constraint problem used to generate the level. The basic location categories are *forward reachable* — possible to reach going forward from the start — and *backward reachable* — possible to reach going backward from the goal.

This categorization is done by using constraints to set up a graph-based "search" using the reachability graph of the level. The search is formulated as a constraint satisfaction problem such that the solution to the problem represents the results of the search. To use a constraint problem in which the number of variables and constraints cannot be changed while solving, the search is set up to a fixed depth, by essentially unrolling the search into a fixed number of *layers*, with each layer representing one iteration of the search. So in the first layer of forward reachability search, only the start location is reachable; in the second layer, locations forward reachable from the start are reachable and all others are not; in the third, locations forward reachable from those reachable in the second layer are reachable but others are not, and so on. Backward reachability search is similar, except edges are followed backward from the goal.

	time		range	
	med.	max.	med.	max.
`driller`				
Prior path reachability	0.17s	0.33s	0.25	0.47
Without softlocks	0.53s	0.82s	0.27	0.42
With softlocks	0.51s	0.61s	0.28	0.41
With softlocks, disc. sinks	0.55s	0.81s	0.26	0.39
Impossible	0.50s	0.68s	0.31	0.46
Application: counts	0.67s	2.24s	0.22	0.32
`slide`				
Prior path reachability	0.18s	0.27s	0.25	0.39
Without softlocks	1.38s	2.05s	0.30	0.44
With softlocks	1.44s	2.48s	0.31	0.44
With softlocks, disc. sinks	2.02s	5.63s	0.29	0.43
Impossible	1.33s	1.57s	0.32	0.48
Application: tricky	23.49s	56.54s	0.31	0.44
`mario`				
Prior path reachability	1.59s	3.30s	0.19	0.28
Without softlocks	7.36s	9.07s	0.20	0.29
With softlocks	14.10s	29.07s	0.21	0.31
With softlocks, disc. sinks	23.86s	32.00s	0.25	0.34
Impossible	7.64s	9.09s	0.21	0.32
Application: repair	6.92s	8.31s	0.00	0.01

Table 1: Level generation times and ranges.

Since the number of layers is fixed in advance, it is necessary to make sure there are enough layers to correctly categorize all locations. For example, if there were not enough layers in the forward search, there may be forward reachable locations in the level that do not get categorized as such. Thus, the categorization constraints include a *saturation* check to make sure there are enough layers. This is done by ensuring that the last two layers are the same — there are no additional locations to add to the categorization.

We also categorize locations in the level as *sinks*. Sinks are based on *hazards*: locations in the level where the player loses and would need to restart. Sinks are either hazards or locations where the player will inevitably make progress toward a hazard and thus lose the level. Thus, the level is effectively lost once the player enters a sink, although it may take a number of moves to reach a hazard. The constraint-based search for sink locations uses the layered approach, though it is carried out differently from the forward and backward reachability search (as described below). In this work, for simplicity, we only consider possible hazard locations along the sides of the level, e.g. the bottom, or all four sides. (As a note, in the process of developing this work, we noticed that it was needed to explicitly account for sinks when generating levels without softlocks; otherwise, sink locations would not be generated: since the player could not reach the goal from them, sinks themselves would be considered softlocks.)

A visualization of the layers used to categorize locations during the generation of a level is shown in Figure 2.

Here we describe the reachability categorization constraint problem setup. To set up the categorization constraints, additional variables are allocated. These are primarily:

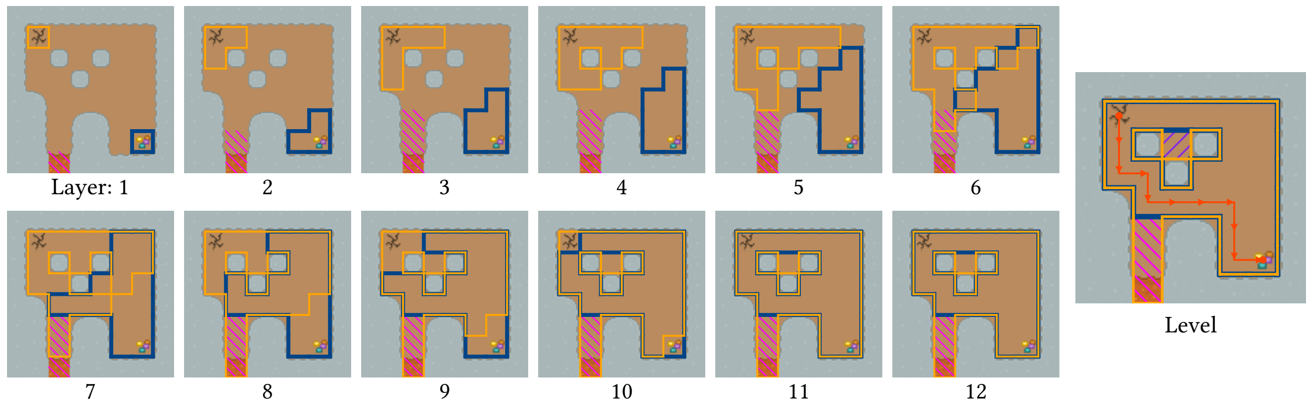

Figure 2: Location categorization from generating a level in the `driller` game, where the player can move side to side and down, but not up. This shows a visualization of the location categorization layers, used internally during constraint solving, to generate a level. These categorizations come from variable values for each layer in the solution to the constraint satisfaction problem. Note that the last two layers are the same due to the saturation constraint. The final generated level shows a path and softlock area. Yellow outlines show forward reachable areas, blue outlines show backward reachable areas, pink backhatching shows sink areas. Red arrows show a path through the level.

- Each node gets a single variable that represents if it is *open*.
- Each edge gets a single variable that represents if it is *traversable*.
- For each layer, each node gets a variable that represents if it is *forward reachable*, a variable that represents if it is *backward reachable* and variable that represents if it is a *sink*.

Other "auxiliary" variables may be used when setting up specific constraints.

The basic node open and edge traversable variables are constrained as:

- A node is open iff the tile at its corresponding location is open.
- An edge is traversable iff the nodes that it needs to be open are open and the nodes that it needs to be closed are not open.

Forward reachability categorization is constrained as:

- In the first layer, a node is forward reachable iff it is the start node.
- In subsequent layers, a node is forward reachable iff: it is itself forward reachable in the previous layer, or it has an *incoming* traversable edge *from* a node that is forward reachable in the previous layer.
- In the final layer, the *goal* node must be forward reachable.
- In the final layer, a node is forward reachable iff it is forward reachable in the previous layer. (This is the saturation constraint.)

Backward reachability (which is essentially forward reachability in reverse) is constrained as:

- In the first layer, a node is backward reachable iff it is the goal node.
- In subsequent layers, a node is backward reachable iff: it is itself backward reachable in the previous layer; or it has an *outgoing* traversable edge *to* a node that is backward reachable in the previous layer.
- In the final layer, the *start* node must be backward reachable.
- In the final layer, a node is backward reachable iff it is backward reachable in the previous layer (saturation constraint).

Sinks are constrained as:

- In the first layer, a node is a sink iff it is a hazard and open.
- In subsequent layers, a node is a sink iff: it is a hazard and open; or, it is a sink in the previous layer; or, if it has any traversable outgoing edge, and all traversable outgoing edges lead to sinks in the previous layer, and it is open.
- In the final layer, a node being a sink implies that it is not the start and it is not the goal.
- In the final layer, a node is a sink iff it is a sink in the previous layer (saturation constraint).

Note that sink categorization only needs to be included if there are possibly any hazards in the level.

With these categorization variables, it is possible to add other specialized constraints relating them to ensure certain level properties.

4 Generating Levels

Here we demonstrate how the constraints described above, categorizing forward and backward reachability and sinks, can be used to produce a variety of effects in levels. For each level generation setup described, 50 levels were generated. A summary of times for generating levels are given in Table 1. Table 1 also shows a summary measure of the *range* of levels generated, where range is based on all pairwise, per-tile differences between levels (e.g. two identical levels would have a difference of 0, and two levels with differing tiles at every location would have a difference of 1).

In this work we used PySAT's [20] RC2 solver [21]. The level images with overlays were created using level2image [8]. Generated levels can be found on OSF at https://osf.io/yr5zb/.

All levels (other than those using path reachability, described below) also apply these constraints to make sure there is at least

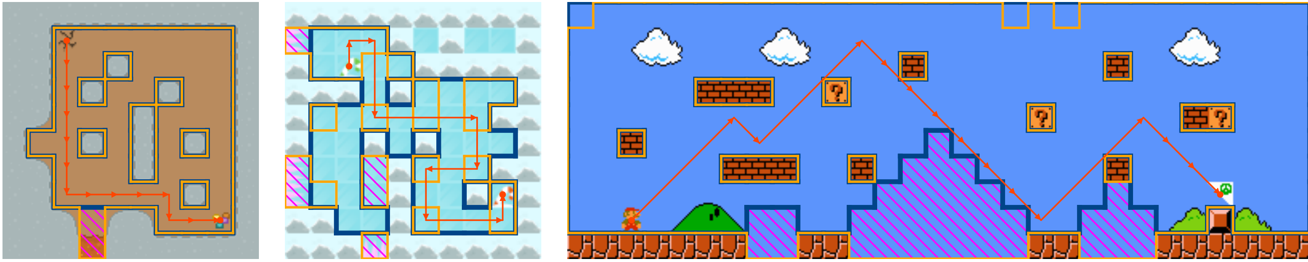

Figure 3: Sample generated levels without softlocks. Yellow outlines show forward reachable areas, blue outlines show backward reachable areas, pink backhatching shows sink areas. Red arrows show a path through the level.

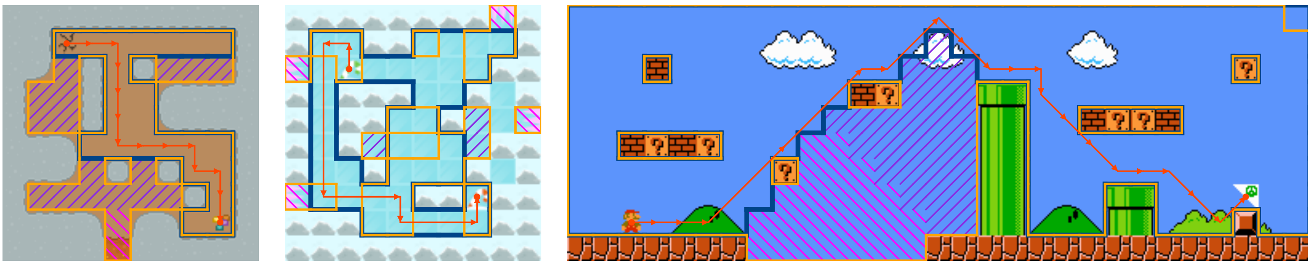

Figure 4: Sample generated levels with softlocks. Yellow outlines show forward reachable areas, blue outlines show backward reachable areas, and pink backhatching shows sink areas, and purple hatching shows softlock areas. Red arrows show a path through the level.

one sink in the level, and that sinks are forward reachable from the start:

- A node being a sink implies that it is forward reachable.
- There is at least one sink.

4.1 Games Used

Here we briefly describe the games used in this work and some parameters of the generated levels. The pattern and reachability templates used are shown in Figure 1.

`driller` is a custom game in which the player drills through the ground. They can move from side to side and down, but not up, with the `driller` reachability template. Tile patterns are learned from a custom example level using the `nbr-2` pattern template; tile images from Kenney [25] were used. Generated levels are 10×10 with the start and goal within 3 tiles of the top-left and bottom-right, respectively. For completable `driller` levels, we apply a constraint so that disconnected sections of the level are not generated:

- A node being open implies that it is forward reachable.

20 reachability steps were used, and the bottom side can be hazards.

`slide` is a custom game where the player slides on ice. They can move up, down, left, and right, but keep moving in the same direction until they run into a solid block. This uses the `slide` reachability template. Tile patterns are learned from a custom example level using the `nbr-2` pattern template; tile images from Kenney [25] were used. Generated levels are 10×10 with the start and goal within 3 tiles of the top-left and bottom-right, respectively. 20 reachability steps were used, and all sides can be hazards.

`mario` is based on the game Super Mario Bros. It is a platforming game where the player can run, jump, and fall. The `mario` reachability template, a discrete approximation of platforming movement similar to that of [40], was used. The example level was based on level 1-1 from the VGLC [41]. Levels for the `mario` game were generated in a two-pass approach: first, the underlying "functional" level is generated using patterns learned by the `ring` template; then, this functional level is used to generate the image for the level using patterns learned by the `nbr-2` template. Generated levels are 10×29 with the start and goal within 4 tiles of the left and right, respectively. 25 reachability steps were used, and the bottom side can be hazards.

4.2 Levels using Prior Path Reachability

As a point of comparison for timing, we generated levels using Sturgeon's previous approach of ensuring that there is a single path through the level using the game's reachability template. More details can be found in [7]. This approach does not ensure that it is not possible to get stuck somewhere. It also does not determine forward or backward reachability or sinks, so it is not possible to use the specialized constraints described above. However, to have some structural similarity with other generated levels, we required at least one of the potential hazard locations for the game (e.g. along the sides) to be open.

4.3 Levels without Softlocks

To generate levels where it is not possible for the player to get stuck, we also apply this specialized constraint to levels:

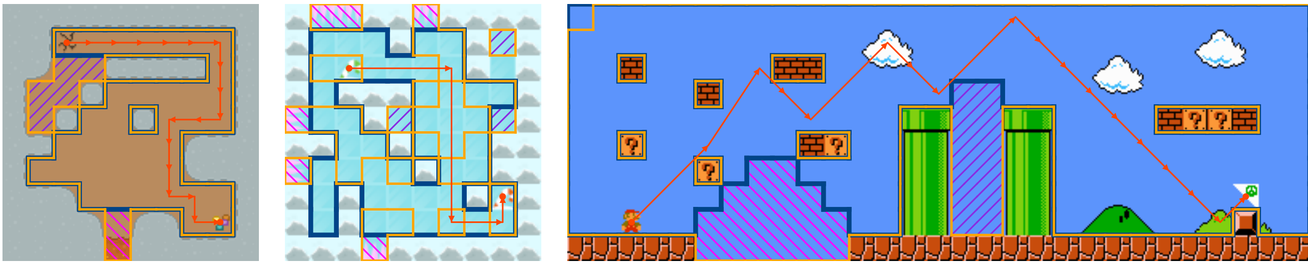

Figure 5: Sample generated levels with softlocks that are disconnected from sinks. Yellow outlines show forward reachable areas, blue outlines show backward reachable areas, and pink backhatching shows sink areas, and purple hatching shows softlock areas. Red arrows show a path through the level.

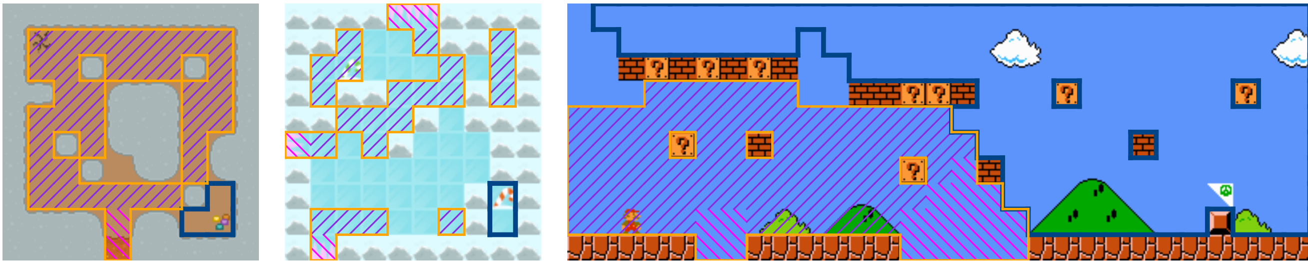

Figure 6: Sample generated levels that are impossible to complete. Yellow outlines show forward reachable areas, blue outlines show backward reachable areas, and pink backhatching shows sink areas, and purple hatching shows softlock areas.

- A node being forward reachable and not a sink implies that it is backward reachable.

This ensures that every location that the player can reach from the start, they can also reach the goal from; or, they will eventually reach a hazard. Thus they won't get stuck. Some sample levels are shown in Figure 3.

4.4 Levels with Softlocks

To generate levels where it is possible for a player to get softlocked, we apply a specialized constraint:

- There is at least one node that is forward reachable, not a sink, and not backward reachable.

Some sample levels are shown in Figure 4.

4.5 Levels with Softlocks, Disconnected Sinks

One property of levels generated with the above softlock-creating constraint, is that although they have an area where the player can get stuck (in that the player cannot reach the goal), it may be possible for the player to reach a sink (and then inevitably a hazard), and thus lose the level. In such a case, the player may not be considered truly stuck, as they can make progress toward some ending of the level, even if it is losing.

Thus, we also explored requiring that softlocks be disconnected from any sink areas, in that they cannot be neighboring. We apply an additional specialized constraint:

- It should be possible to get softlocked (as above).
- A node being softlocked (as defined above) implies that all outgoing edges are either not traversable or do not lead to a sink.

Some sample levels are shown in Figure 5.

4.6 Impossible Levels

As noted in previous work, it may be interesting to generate levels that are impossible to complete, for example, to better understand a generator, or what level features are used to block the player from making progress [9, 12]. Levels that are impossible to beat might also be considered ones in which the player starts out stuck.

To generate impossible levels, we flip the final layer reachability constraints above for both forward and backward reachability. The forward reachability constraint is replaced with:

- In the final layer, the *goal* node must **not** be forward reachable.

and the backward reachability constraint is replaced with:

- In the final layer, the *start* node must **not** be backward reachable.

Note that for any impossible levels, `driller` does not use the above-mentioned constraint that all open locations are forward reachable, since there will necessarily be some open areas that are not forward reachable. Some sample levels are shown in Figure 6.

4.7 Generation Times and Ranges

Here we summarize some general comparisons to using prior path reachability as a baseline, before proceeding to game-specific applications.

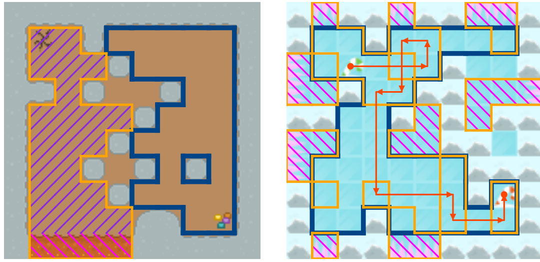

Figure 7: Applications in `driller`, where an impossible level has a balanced amount of its tiles forward and backward reachable; and `slide`, where there are a relatively large number of sinks in the level interior.

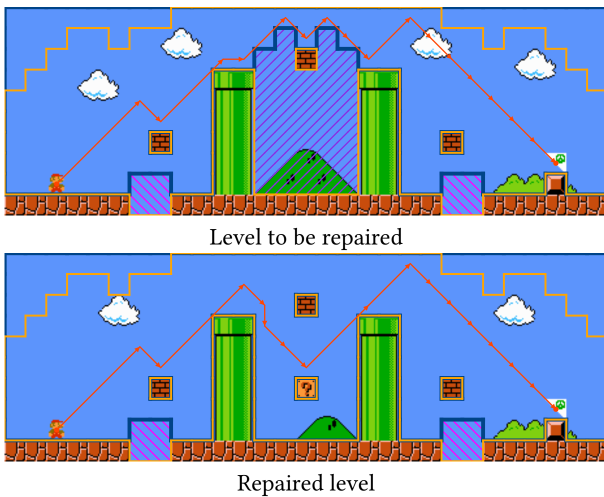

Figure 8: *Repair* application in `mario`. The level to be repaired is shown on the top. Note that is is possible to get stuck in between the pipes. The repaired level has added a block in between the pipes, making it possible to jump out.

Compared to prior path reachability, generating levels with the reachability categorization constraints is notably slower, though the impact depends on the game. From Table 1, generating levels without softlocks takes roughly 3–5 times longer on average, depending on the game. Generating levels with softlocks takes roughly 3–8 times as long, and with the additional constraint that softlocks are disconnected from sinks roughly 3–15 times as long. Generating impossible levels takes roughly 3–5 times as long, similar to levels without softlocks.

Looking at range, with path reachability as a baseline, the average range of levels generated appears to increase slightly, although the maximum range for `driller` decreases.

4.8 Levels with Specific Applications

To highlight the constraint-based approach's flexibility and controllability, we further explored specific applications in each game.

In `driller`, we looked at controlling the number of forward and backward reachable locations in an impossible level. In previous work [9], it was possible to generate an impossible level. However, how much of the level was reachable before the goal was blocked was not controllable and fairly variable. With the current system, it is possible to control the counts of reachability of tiles. We generated levels using specialized constraints:

- The level must be impossible (as above).
- Over a quarter of the tiles must be forward reachable.
- Over a quarter of the tiles must be backward reachable.

These levels take about 4 times as long to generate, on average, as using path reachability. An example is shown in Figure 7.

In `slide`, we looked at making "tricky" levels, where there are a large number of sinks that will lead to hazards. These could be interesting and challenging levels to play, as there are a larger number of locations which will eventually lead to hazards, although this may not be obvious.

- It must not be possible to get stuck in the level (as above).
- At least 8 of the tiles not along the side of the level must be sinks.

These levels take quite a bit longer to generate, roughly 130 times as long as using path reachability. An example is shown in Figure 7.

In `mario`, we look at using this approach to *repair* a level. The system takes as input a level where is is possible to get stuck, and makes minimal changes so that it is no longer possible to get stuck.

- It must not be possible to get stuck in the level (as above).
- [Soft constraint] Each tile in the generated level should have the same "functionality" as the level to be repaired (the appearance of background tiles is not considered).

We used a level where the player can get stuck between two pipes as the input level. Repairing this level turns out to be a bit faster than generating a level from scratch with no softlocks. However, this may be due to the level used requiring only one tile to be changed, and might take longer for levels requiring more changes. The range of levels generated is also very low, as the levels will mostly be the same as the input level. An example is shown in Figure 8.

We also note that by omitting the constraint that it is not possible to get stuck, and making the tile matching constraint hard, it is possible to analyze the reachability in an existing level to e.g. find softlocks (although, in this case it was necessary to use 30 layers).

5 Conclusion

In this work we presented a constraint-based approach to generating 2D tile-based levels without softlocks. The approach uses constraints to categorize locations as forward reachable, backward reachable, or sinks. Using this categorization, additional constraints can be added that prevent softlocks. We applied the approach in three games with different player movement. We also showed how the location categorization can be used to create other effects in levels, including generating levels with softlocks, generating impossible levels, and repairing levels. We found that this approach is notably slower than generating levels when only ensuring there is a path from the start to the goal.

There are several avenues for future work. Here, we only considered player movement. However, softlocks are often of interest in games with inventory or powerup systems, such as Super Metroid, where player movement or accessible areas can change over the

course of gameplay. It would also be interesting to explore softlock prevention in other genres of games that are not based on a single player's navigation, or in domains such as interactive fiction. Additionally, further characterization of the prevalence of softlocks in generated levels could help inform when approaches to prevent them would be most useful. Specific applications, such as level repair, could be explored more thoroughly.

Acknowledgments

The authors would like to thank Adam Smith and Chris Martens for helpful discussions related to this work.

References

[1] Parosh Aziz Abdulla, Bengt Jonsson, Ahmed Rezine, and Mayank Saksena. 2006. Proving liveness by backwards reachability. In *CONCUR 2006 – Concurrency Theory*, Christel Baier and Holger Hermanns (Eds.). Springer, Berlin, Heidelberg, 95–109. doi:10.1007/11817949_7

[2] Aghyad Albaghajati and Moataz Ahmed. 2022. A co-evolutionary genetic algorithms approach to detect video game bugs. *Journal of Systems and Software* 188 (June 2022), 111261. doi:10.1016/j.jss.2022.111261

[3] Bowen Alpern and Fred B. Schneider. 1985. Defining liveness. *Inform. Process. Lett.* 21, 4 (Oct. 1985), 181–185. doi:10.1016/0020-0190(85)90056-0

[4] Joakim Bergdahl, Camilo Gordillo, Konrad Tollmar, and Linus Gisslén. 2020. Augmenting automated game testing with deep reinforcement learning. In *2020 IEEE Conference on Games (CoG)*. 600–603. doi:10.1109/CoG47356.2020.9231552

[5] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded model checking using satisfiability solving. *Formal Methods in System Design* 19, 1 (July 2001), 7–34. doi:10.1023/A:1011276507260

[6] Michael Cook and Azalea Raad. 2019. Hyperstate space graphs for automated game analysis. In *2019 IEEE Conference on Games (CoG)*. 1–8. doi:10.1109/CIG.2019.8848026

[7] 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.

[8] Seth Cooper. 2024. level2image: a utility for making 2D tile-based level images with overlays. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 20, 1 (Nov. 2024), 254–255. doi:10.1609/aiide.v20i1.31886

[9] Seth Cooper and Mahsa Bazzaz. 2024. Literally Unplayable: On Constraint-Based Generation of Uncompletable Levels. In *Proceedings of the 15th Workshop on Procedural Content Generation*.

[10] Seth Cooper and Anurag Sarkar. 2020. Pathfinding Agents for Platformer Level Repair. In *Proceedings of the Experimental AI in Games Workshop*.

[11] Riccardo Coppola, Tommaso Fulcini, and Francesco Strada. 2024. Know Your Bugs: A Survey of Issues in Automated Game Testing Literature. In *2024 IEEE Gaming, Entertainment, and Media Conference (GEM)*. 1–6. doi:10.1109/GEM61861.2024.10585558

[12] 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.

[13] Jose M. Font, Roberto Izquierdo, Daniel Manrique, and Julian Togelius. 2016. Constrained Level Generation through Grammar-Based Evolutionary Algorithms. In *Applications of Evolutionary Computation (Lecture Notes in Computer Science)*, Giovanni Squillero and Paolo Burelli (Eds.). Springer International Publishing, Cham, 558–573.

[14] Jonas Gillberg, Joakim Bergdahl, Alessandro Sestini, Andrew Eakins, and Linus Gisslén. 2023. Technical challenges of deploying reinforcement learning agents for game testing in AAA games. In *2023 IEEE Conference on Games (CoG)*. 1–8. doi:10.1109/CoG57401.2023.10333194

[15] Camilo Gordillo, Joakim Bergdahl, Konrad Tollmar, and Linus Gisslén. 2021. Improving playtesting coverage via curiosity driven reinforcement learning agents. In *2021 IEEE Conference on Games (CoG)*. 1–8. doi:10.1109/CoG52621.2021.9619048

[16] Michael X. Grey, Caelan R. Garrett, C. Karen Liu, Aaron D. Ames, and Andrea L. Thomaz. 2016. Humanoid manipulation planning using backward-forward search. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 5467–5473. doi:10.1109/IROS.2016.7759804

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

[18] Christoffer Holmgård, Michael Cerny Green, Antonios Liapis, and Julian Togelius. 2019. Automated playtesting with procedural personas through MCTS with evolved heuristics. *IEEE Transactions on Games* 11, 4 (Dec. 2019), 352–362. doi:10.1109/TG.2018.2808198

[19] Gwan-Hwan Hwang, Kuo-Chung Tai, and Ting-Lu Huang. 1995. Reachability testing: an approach to testing concurrent software. *International Journal of Software Engineering and Knowledge Engineering* 05, 04 (Dec. 1995), 493–510. doi:10.1142/S0218194095000241

[20] 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.

[21] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. 2019. RC2: an efficient MaxSAT solver. *Journal on Satisfiability, Boolean Modeling and Computation* 11, 1 (Jan. 2019), 53–64. doi:10.3233/SAT190116

[22] Mark R. Johnson. 2019. Playful work and laborious play in Super Mario Maker. *Digital Culture & Society* 5, 2 (Dec. 2019), 103–120. doi:10.14361/dcs-2019-0207

[23] 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*. 1–10.

[24] Jediah Katz, Bahar Bateni, and Adam M. Smith. 2024. You-only-randomize-once: shaping statistical properties in constraint-based PCG. In *Proceedings of the 19th International Conference on the Foundations of Digital Games*. 1–11.

[25] Kenney. 2010. Home. https://www.kenney.nl/.

[26] Lazaros Lazaridis and George F. Fragulis. 2024. Creating a newer and improved procedural content generation (PCG) algorithm with minimal human intervention for computer gaming development. *Computers* 13, 11 (Nov. 2024), 304. doi:10.3390/computers13110304

[27] A. Ligęza. 1985. Backward versus forward plan generation. *IFAC Proceedings Volumes* 18, 16 (Nov. 1985), 337–342. doi:10.1016/S1474-6670(17)59986-3

[28] 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 (FDG'21)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/3472538.3472542

[29] Ian M. Mitchell. 2007. Comparing forward and backward reachability as tools for safety analysis. In *Hybrid Systems: Computation and Control*, Alberto Bemporad, Antonio Bicchi, and Giorgio Buttazzo (Eds.). Springer, Berlin, Heidelberg, 428–443. doi:10.1007/978-3-540-71493-4_34

[30] 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.

[31] Kazuhiro Ogata and Kokichi Futatsugi. 2010. A combination of forward and backward reachability analysis methods. In *Formal Methods and Software Engineering*, Jin Song Dong and Huibiao Zhu (Eds.). Springer, Berlin, Heidelberg, 501–517.

[32] Andrew Plotkin. 1996. The Zarfian Cruelty (or Forgiveness) Scale. https://eblong.com/zarf/essays/cruelty.html. Accessed: 2025-01-30.

[33] Christian Reuter, Stefan Göbel, and Ralf Steinmetz. 2015. Detecting structural errors in scene-based Multiplayer Games using automatically generated Petri Nets. In *Proceedings of the 10th International Conference on the Foundations of Digital Games*.

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

[35] 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*. 156–163.

[36] 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.

[37] 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.

[38] Christian Stangier and Thomas Sidle. 2004. Invariant checking combining forward and backward traversal. In *Formal Methods in Computer-Aided Design*, Alan J. Hu and Andrew K. Martin (Eds.). Springer, Berlin, Heidelberg, 414–429. doi:10.1007/978-3-540-30494-4_29

[39] 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.

[40] Adam James Summerville, Shweta Philip, and Michael Mateas. 2015. MCMCTS PCG 4 SMB: Monte Carlo Tree Search to Guide Platformer Level Generation. *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment; Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference* (2015).

[41] 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

[42] Richard N. Taylor. 1983. A general-purpose algorithm for analyzing concurrent programs. *Commun. ACM* 26, 5 (May 1983), 361–376. doi:10.1145/69586.69587

[43] 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.

[44] Yakir Vizel, Orna Grumberg, and Sharon Shoham. 2013. Intertwined forward-backward reachability analysis using interpolants. In *Tools and Algorithms for the Construction and Analysis of Systems*, Nir Piterman and Scott A. Smolka (Eds.). Springer, Berlin, Heidelberg, 308–323. doi:10.1007/978-3-642-36742-7_22

[45] Benedict Wilkins and Kostas Stathis. 2022. World of Bugs: a platform for automated bug detection in 3D video games. In *2022 IEEE Conference on Games (CoG)*. 520–523. doi:10.1109/CoG51982.2022.9893616

[46] 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.