Proceedings of the Twenty-First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2025)

Spacetime Level Generation and Editing with Constraints from Examples

Akshar Vandara, Kaylah Facey, Seth Cooper

Northeastern University, Boston, MA, USA
{vandara.a, facey.k, se.cooper}@northeastern.edu

Abstract

When generating levels for tile- and turn-based 2D games, generating a solution along with the level can ensure that the level is completable. One approach is to represent the level and solution as a 3D spacetime block with two spatial dimensions and one time dimension. Recent work in Space-Time WaveFunctionCollapse (STWFC) explored learning to generate such level-solution blocks from example blocks by adapting 3D WaveFunctionCollapse for two spatial and one temporal dimension. While this generated solvable levels, the approach was found to be slow and could violate global constraints such as having a single player. Thus, in this work we explore learning to generate spacetime level-solution blocks from examples by applying an existing 2D constraint-based level generation system. Rather than extending the 2D generator to 3D, we encode the time dimension in 2D by using a filmstrip representation where each frame represents a step in the time dimension. We compare STWFC and two filmstrip-based approaches: block constraints across frames that are analogous to those of STWFC, and constraints based on differences between frames. We demonstrate an application for interactive spacetime editing of levels and their solutions in the filmstrip representation, along with generating variations of levels sharing the same solution path, and variations of levels whose solution goes through the same arrangement of tiles at a point in time.

Introduction

Procedural content generation (PCG) has long been used in games to create content such as levels in automated or semi-automated ways (Shaker, Togelius, and Nelson 2016). More recently, machine learning approaches that can learn from example content have been integrated as PCGML (Summerville et al. 2018). PCGML can be used to generate various types of content, including levels. One challenge in PCG and PCGML when applied to levels is ensuring that the generated level is actually completable. It would be undesirable to generate levels for which there is no solution.

One particular PCGML approach, WaveFunctionCollapse (Gumin 2016), has gained popularity by learning local patterns of tile arrangements in 2D levels and then generating new levels consisting of those patterns. While this approach can capture the style of training levels, it doesn't ensure completability of generated levels. We recently introduced Space-Time WaveFunctionCollapse (STWFC)—a method of encoding a 2D level solution as a 3D grid of stacked *frames* (timesteps in the solution) $row \times col \times frame$, making a spacetime level-solution block (Facey and Cooper 2024). This block was then passed into an off-the-shelf WaveFunctionCollapse solver to generate a level with its solution.

While the STWFC method could generate completable levels, it was found to be slow even for very small levels,

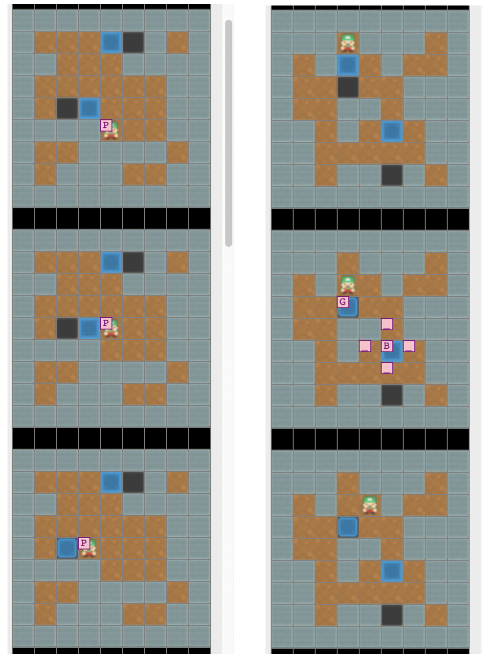

Figure 1: Screenshot of spacetime editing in Sokoban. The time dimension is unrolled into a filmstrip representation, with each frame representing a step in the solution. Spacetime constraints between frames are learned from a training filmstrip, and tile constraints added by the user are shown as a purple overlay. On the left, a solution path for the player across all frames is added, and on the right, a desired arrangement of tiles in a specific frame is added. A filmstrip satisfying the constraints is generated.

Copyright © 2025, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

and would sometimes violate global constraints such as only having one player in the level.

Therefore, in this work we further explore spacetime level generation from examples and compare to STWFC. In particular, we use the Sturgeon constraint-based level generation system (Cooper 2022b) and two methods of learning and applying patterns: the *block method*, which uses fixed-shape blocks similar to STWFC, and the *diff method*, which uses dynamic pattern shapes based on changes between neighboring frames.

As Sturgeon generally works on 2D levels, rather than extending it to support 3D spacetime blocks, we encode 3D spacetime blocks into a 2D *filmstrip* representation by unrolling frames along the row-axis of the grid. This allows Sturgeon to generate filmstrips with only a few modifications to pattern handling in the core generator of the system. These include organizing parts of patterns into groups and handling of the specialized void tiles that surround the grid and separate frames in the filmstrip. The 2D filmstrip representation may provide opportunities for other existing 2D PCG or PCGML approaches to extend to spacetime.

Based on the new approach, we also developed a simple mixed-initiative spacetime level editor that visualizes the generated filmstrip and allows the user to overlay additional tile constraints at any point in spacetime. This allows the user to edit not just the starting level (i.e. the first frame of the filmstrip), but any part of a desired solution in spacetime. We present this editor mainly as proof-of-concept and do not thoroughly evaluate it.

We found that the Sturgeon-based methods are remarkably faster than STWFC, over an order of magnitude. We also demonstrate how spacetime editing can work in the puzzle games Sokoban and Block Dude. Generated levels and videos of interactions with the spacetime editing application can be found at https://osf.io/9xspu/.

Related Work

Although distantly related, this work takes inspiration from research in character animation. In the animation domain, spacetime constraints were proposed as a way to allow animators to specify e.g. higher level specifications on a character's motion, and allow physical constraints to solve for the animation (Witkin and Kass 1988). Motions can also be edited by e.g. changing physical properties and resulting in physically accurate motions (Liu, Hertzmann, and Popović 2005). In much the same way we envision that game designers might spacetime edit levels they are working on, matching their constraints while ensuring the levels are still completable.

Much work in PCG considers ensuring the completability of generated levels. A common approach is "generate-and-test", where levels are generated and then in a separate step tested by a gameplay agent to ensure they can be completed, or have completability factored into an objective function (Togelius et al. 2011; Snodgrass and Ontañón 2014; Volz et al. 2018; Withington 2020; Summerville and Mateas 2016; Snodgrass and Ontanon 2015). Some search-based approaches use level completability to divide levels into feasible and infeasible such as with the FI-2Pop algorithm (Viana et al. 2022). Other work has sought to repair generated levels that may be uncompletable, rather than discard them and start over. This includes approaches using agents that can modify the level (Cooper and Sarkar 2020) as well as machine learning (Jain et al. 2016; Shu et al. 2020) and constraint-based approaches (Zhang et al. 2020).

Several PCGML approaches to learning tile patterns from examples and using them to generate levels have been developed. The WaveFunctionCollapse algorithm mentioned earlier has seen popularity in research (Gumin 2016), as well as extensions and variations on the basic approach (Karth and Smith 2017, 2019; Kim et al. 2020a), although these are generally 2D. N-grams are another related, essentially 1D approach, to learning patterns from examples (Dahlskog, Togelius, and Nelson 2014; Summerville, Philip, and Mateas 2015).

There have been several approaches to learning game mechanics (e.g., how the game state changes from frame to frame) from examples. We build on our work on *STWFC* (Facey and Cooper 2024), which used a 3D spacetime grid and an off-the-shelf WFC solver to successfully capture game mechanics. Other systems for learning mechanics from examples include BlockStudio (Banerjee et al. 2016) and Mechanic Maker (Sumner, Saini, and Guzdial 2024); our diff method of learning changes between frames can be considered similar to these. Other work has explored learning game mechanics or engines directly from video of gameplay (Guzdial, Li, and Riedl 2017). Approaches related to world models (Ha and Schmidhuber 2018) have been used to model games (Kim et al. 2020b) and even generate new games (Bruce et al. 2024; Yu et al. 2025) at the pixel level.

As the diff method we present learns tile pattern differences between frames, it could also be considered a form of learning tile rewrites. There is much previous work in representing games and other dynamic systems with tile rewrites, ranging from early work in the BITPICT animation system (Furnas 1991) to more recent work such as MarkovJunior (Gumin 2022). Closely related to our current work is Sturgeon-MKIII (Cooper 2023), which also uses the Sturgeon system and tile rewrites to generate level playthroughs. However, Sturgeon-MKIII requires manual authoring of possible rewrites rather than learning them, and explicitly models the time dimension rather than encoding it in a filmstrip.

There have also been a number of mixed-initiative level editors, in which the user and the editor system can take turns editing the level or making suggestions (Yannakakis, Liapis, and Alexopoulos 2014). Most closely related to our work is mixed-initiative Wave Function Collapse (miWFC) (Langendam and Bidarra 2022), which allows interactive control of WFC, such as manually drawing tiles on the level. However, as with most WFC-related work, miWFC looks at the initial level (essentially the first frame) rather than the whole spacetime solution. Recently, the game TownScaper (Stålberg 2021) allows the player to interactively build a town using WFC-like adjacency rules.

Other work has explored mixed-initiative level editors that can suggest variations to the user, such as Sentient Sketch-

Figure 2: Overview of applying the block and diff methods on the simple demonstration game. In the learning process, given the training filmstrip, patterns are learned. Input-only patterns show tiles with solid blue outlines; input-output patterns show the input tiles lighter with dotted blue outlines and output tiles with solid purple outlines. In the generation process, learned patterns are applied to generate a new filmstrip of a longer level.

book (Liapis, Yannakakis, and Togelius 2013) and Evolutionary Dungeon Designer (Alvarez et al. 2018), or learn from the user as they edit a level, such as Morai Maker (Guzdial et al. 2019).

System Overview

Sturgeon

This work uses Sturgeon (Cooper 2022b), a system for using constraint solvers to generate tile-based levels. A *tile* has a text character (and possibly image) associated with it, and tiles are arranged on a 2D grid of locations. Sturgeon sets up level generation as a Boolean constraint satisfaction problem. Roughly, there is a variable for each possible tile at each location, and the values of these variables are constrained by higher-level design rules converted to Boolean constraints. Such rules include having exactly one tile at each location, and that local arrangements of tiles—referred to as *patterns*—match those learned from example training levels. This problem is then solved by a lower-level (e.g. SAT) constraint solver, and the solution (if found) is interpreted as the tiles for the generated level. Sturgeon's code is available at https://github.com/crowdgames/sturgeon-pub, and code used to create this work is available at https://github.com/crowdgames/sturgeon-eval-spacetime.

In this work, we re-purpose a number of existing Sturgeon features to encode spacetime (two spatial and one temporal) playthroughs of a level into a 2D *filmstrip* representation, with each *frame* representing a step in the playthrough. The filmstrip representation (shown, for example, on the left of Figure 2) unrolls all frames in the playthrough of a level along the row dimension, and separates them with a gap of specialized *void* tiles, which are considered to be outside the level (any location outside the grid dimensions is also considered to contain a void tile). For the purposes of this work, void tiles are ignored when considering patterns. The first frame of the filmstrip is the generated level, and the remaining frames show how the level can be completed.

Tile patterns in Sturgeon are fairly flexible in terms of the arrangement of tiles and do not necessarily need to be contiguous. Patterns consist of an *input* component and, optionally, *output* components. Patterns with both input and output components create an implication constraint: the input pattern being present at a location in the generated level implies that one of its output patterns is also present at a relative location. Patterns with only an input component simply allow that pattern to be present in the generated level. Every location in the generated level has to correspond to some pattern's input component.

Sturgeon also supports various configuration grids that can be used to configure the constraint problem to be solved. First, a *tag* grid, which specifies which groups of tiles can be placed at locations in the grid. In this work, the tag grid is used to specify the filmstrip layout for a generated level by requiring void tiles between the frames. Second, a *match* grid, which specifies specific tiles that must be placed at particular locations. In this work, we use the match grid to require specific padding tiles around the edge of the level. Third, a *pool* grid, which can be used to pool learned and applied patterns into separate groups, based on their location in the grid. The pool grid was referred to as the *game* grid in previous work (Cooper 2022a), though in this work we use this grid to specify the patterns learned and applied at different frames within the same game, and so use a different name for it. (In cases where these grids are not provided, they use reasonable default values.)

In this work, we explored two different methods for learning and applying spacetime patterns in the filmstrip representation (discussed further below). First, the *block* method, which is intended to be as similar as possible to the STWFC (Facey and Cooper 2024) method, and uses cuboid block input-only patterns. Second, the *diff* method, which is centered around dynamically learning input-output patterns based on what tiles change between frames.

For the most part, this work made use of existing Sturgeon features as described in the filmstrip representation. The primary modification to the core generator was to the handling of input-output patterns in the diff method implementation. The system pattern handling was updated to allow organizing output patterns into groups and constraining them independently in the generator (described more below). The system was also updated to disregard patterns with void tiles, in addition to various utilities for remapping patterns (also described more below) and e.g. merging individual frames into a filmstrip.

To solve the constraint problems, in this work we used PySAT's (Ignatiev, Morgado, and Marques-Silva 2018) gluecard41 SAT solver, which according to documentation is based on the Glucose (Audemard and Simon 2009), MiniCARD (Liffiton and Maglalang 2012), and MiniSAT (Eén and Sörensson 2003) solvers.

Here, to provide an overview of each method, we use a simple demonstration game (shown in Figure 2) where the player (`P`) can only move right through empty space (`_`) to reach the door (`D`) and enter it, achieving the goal (`G`).

Block Method

The block method is illustrated on the top of Figure 2.

The top-left illustrates the pattern learning process. A training filmstrip is provided. Void tiles separating frames are solid black; here there is one row of gap between each frame. As in STWFC (Facey and Cooper 2024): the last few frames are duplicated, which allows them to be repeated so the generated filmstrip can end early; the sides are padded with blank tiles; and the final frame of the filmstrip is a special *terminal* (`T`) tile so that the penultimate frame above it will have completed the level. In this case, the equivalent of $1row \times 2col \times 2frame$ block input-only patterns are learned. In the filmstrip representation, the frame part of the pattern is also unrolled along the row axis, by the amount of the void gap between frames. Since there is one row of gap, the two rows in the pattern have one row between them. This can be considered as the tiles in each pattern corresponding to the 2D (row, col) indices of $(0,0),(0,1),(2,0),(2,1)$, analogous to the 3D $(row, col, frame)$ indices of $(0,0,0),(0,1,0),(0,0,1),(0,1,1)$. These patterns are learned simply by scanning the level and keeping all unique patterns found. There is no training pool grid, so all learned patterns go into the same pool. Note that the patterns with terminal tiles on the bottom only have blank and goal tiles above them which ensures level completion in the penultimate frame.

The top-right illustrates the configuration to generate a filmstrip that is wider than the original. Additional custom constraints can be provided; in this case that there is one player starting on the left and one door starting on the right. A tag grid specifies the frame size and number of frames in the generated filmstrip; the comma (`,`) tag is a default specifying any tile can be placed. A match grid specifies that the sides should be padded with blanks and the final frame should be all terminal tiles; the comma (`,`) is a wildcard match allowing any tile. The pattern constraints are applied by scanning across the filmstrip to be generated, and with this configuration, the generated level is shown on the top far-right.

Diff Method

The diff method is illustrated on the bottom of Figure 2.

The bottom-left illustrates the pattern learning process. In this case, the training filmstrip does not have the frame of terminal tiles. A training pool grid is provided that splits the filmstrip into three pools: pools 0, 1, and 2, which will contain patterns respectively for the initial frame, intermediate frames, and final frame. In this case, pools 0 and 2 use basic $1row \times 2col$ block input-only patterns to capture local tile arrangements of how the level looks at the start and when it is completed. Note that pool 2 only has patterns with blanks and goals, which shows level completion in the final frame.

Pool 1, for intermediate frames, uses a frame diff approach to learning input-output patterns by comparing adjacent frames. The diff method depends on a modification to how input-output patterns are handled in Sturgeon. By default, and in previous work, input-output patterns are applied such that an input pattern being present implies that *at least one associated output* is present. In this work, we allow output patterns to be further organized into *groups*. When applying these grouped diff patterns as constraints, an input pattern being present implies that *exactly one associated output from each group* is present.

Thus, pool 1 patterns are split into two groups, a *prev* group, and a *next* group, depending on which frame is compared to. For the prev group, each frame is compared tile-by-tile to the previous frame. The tile in the current frame is considered the input pattern. If the corresponding tile in the previous frame is the same, then a pattern with just the previous tile at its relative location as an output is learned. This kind of pattern allows tiles to stay in the same place across frames. If the previous tile is different, then a pattern with all the tiles that changed between the two frames (other than the current tile) at their relative locations is learned. The patterns for the next group are learned analogously by comparing with the next frame.

The practical purpose of using both use both prev and next groups, instead of just one or the other, is to connect the first intermediate frame to the initial frame through prev, and the last intermediate frame to the final frame through next. Other intermediate frames will end up being connected to both neighboring frames.

Consider, for example, the learned diff patterns associated with the player tile of the demonstration game in Figure 2. There are two next and one prev pattern learned. In an input-output pattern, the input pattern being present will imply an output pattern in each group. The two next group patterns

Game	Method	Time (s)	Eff. Length	Success	Trivial	Extra P
Field	STWFC	10.793 (8.457)	3.2 (1.6)	0.5	0.6	0.6
	Block	0.335 (0.008)	2.7 (2.1)	1.0	0.6	0.0
	Diff	0.049 (0.003)	5.0 (2.0)	1.0	0.2	0.0
Maze	STWFC	56.665 (44.788)	3.5 (1.9)	0.6	0.5	0.0
	Block	1.523 (0.053)	1.0 (0.0)	1.0	1.0	0.0
	Diff	0.130 (0.000)	1.0 (0.0)	1.0	1.0	0.0
Sokoban-Small	STWFC	79.756 (66.790)	3.83 (1.67)	0.6	0.5	0.5
	Block	0.949 (0.044)	3.4 (1.56)	1.0	0.5	0.0
	Diff	0.141 (0.005)	5.5 (0.67)	1.0	0.0	0.0

Table 1: Comparison generation summary across games and methods. Means and standard deviations or proportions given.

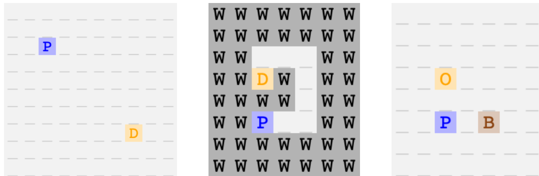

Figure 3: First frame of the 3 comparison training examples, including padding around the frame. From left to right: *Field 1*, *Maze 1*, *Soko 1*.

represent the next frame, the player has to move onto either a blank, leaving a blank where they were, or onto a door, leaving a blank where they were and changing to a goal. The single prev group pattern represents that in the previous frame, the player must have moved into a blank and left a blank behind. Note there is no pattern with the player staying still, so they have to move each frame.

The bottom-right illustrates the level generation process. It is fairly similar to the block method, except that there is no last frame with all terminal tiles and a pool grid is used to specify where patterns from each pool are to be applied. Additionally, the modification to pattern applications to require exactly one output pattern from each group is used. Practically speaking, this modification was found to prevent, for example, a player splitting into multiple players traveling in different directions. With this configuration, the generated level is shown on the far bottom-right.

Pattern Row Remapping

Notably, in the filmstrip representation, all frames are expected to be the same height. When learning from or generating levels of the same height but different widths, this does not cause an issue.

However, what if we want to learn patterns from, and generate, filmstrips with varying heights (e.g. learn from filmstrips with frame heights 7 and 9, and generate with frame height 8)? We accomplish this by remapping the rows of learned patterns; that is, change the row coordinates, if needed, in the patterns so that they correspond to what they would be for a filmstrip of a different height. With multiple training filmstrips, we select a preferred frame height (usually that of the tallest training frames, e.g. height 9), remap all learned patterns to this preferred height, and then merge them together. To generate a filmstrip with frames a different height from the preferred height, we first remap the patterns from the preferred height to the desired frame height (e.g. height 8), and then use these remapped patterns to generate the filmstrip.

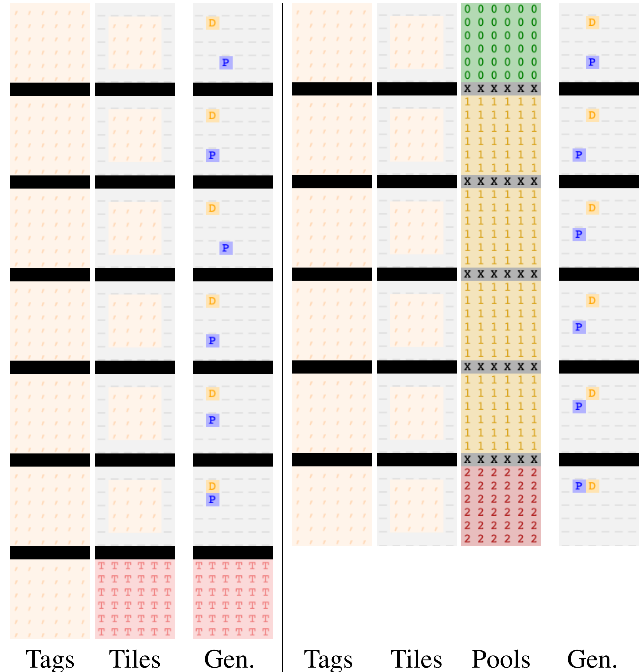

Figure 4: Generating a Field level using the block method (left) and diff method (right).

Comparison to STWFC

Method

To compare with the previous approach, we compared STWFC to the block and diff methods on the three games used in the STWFC work. Though Facey and Cooper (2024) used two training examples for each game, we used only the smaller example for each game (see Figure 3), which as in Facey and Cooper (2024) was rotated and flipped to increase the number of potential examples.

Field — The player navigates an open field to reach a door. We trained with *Field 1* from Facey and Cooper (2024), which has effective size $8row \times 8col \times 12frame$, padded

Game	Time (s)	Eff. Length	Density
Sokoban-Large $7 \times 7 \times 10$	0.918 (0.511)	9.400 (0.800)	0.742 (0.050)
Sokoban-large $10 \times 10 \times 20$	11.420 (10.807)	20.000 (0.000)	0.621 (0.022)
Block Dude $4 \times 10 \times 15$	1.100 (0.029)	14.800 (0.400)	0.625 (0.076)
Block Dude $4 \times 16 \times 24$	3.248 (0.550)	23.700 (0.458)	0.537 (0.056)

Table 2: Summary of larger generated levels (using the diff method). Mean (standard deviation) or proportion given.

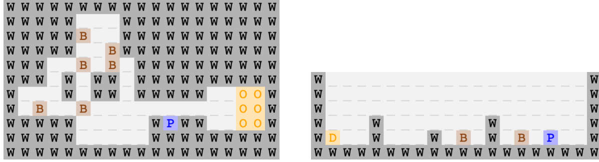

Figure 5: First frame of the training filmstrip for Sokoban-Large (left) and Block Dude (right).

with blanks.

Maze — The player navigates a maze with walls to reach a door. We trained with *Maze 1* from Facey and Cooper (2024), which has effective size $6row \times 5col \times 10frame$, padded with walls.

Sokoban-Small — The player must push a crate onto a target. We trained with *Soko 1* from Facey and Cooper (2024), which has effective size $6row \times 5col \times 12frame$, padded with blanks. *Soko 1* doesn't include any walls to avoid; though the other example we used in Facey and Cooper (2024) does include walls, we found that no walls were generated by STWFC.

The block method was implemented to be as similar to STWFC as possible. For comparisons, both STWFC and the block method used the block size of $3row \times 3col \times 2frame$ used in previous STWFC work. Thus, a main comparison between STWFC and the block method is the underlying use of an off-the-shelf 3D WFC solver versus a SAT solver.

Also to be more similar to STWFC, for comparisons, the diff method learned $3row \times 3col$ patterns for pool 0 from *all* frames, not just the first one, while pool 2 only learned these patterns from the final frame. This was done because the original work uses padding to demarcate a final timestep but not a starting one, considering all timesteps for training examples as equally valid starting timesteps.

The block and diff methods also had additional constraints enforcing that there was only one player, door, block, and target (as appropriate for the game) in the first frame.

For all three games and all three methods (block, diff, and STWFC), we generated ten levels with an *effective size* of $4row \times 4col \times 6frame$, although with padding the *actual size* of the levels were $6row \times 6col$ and either $6frame$ or $7frame$, depending on the method. We computed the same metrics used by Facey and Cooper (2024) — time to generate, effective length of the filmstrip (i.e. number of frames until the level is completed), whether the generation process was successful (i.e. if a solution was found), whether the level was trivial (i.e. it is solved by the third frame), and if there is ever an extra player (`P`) appearing in the level.

A summary of the results is given in Table 1. We also show the generation process for the block and diff methods — just for Field, as the other games are similar — in Figure 4. Images of filmstrips in figures were created using level2image (Cooper 2024).

Discussion

We find that both Sturgeon-based methods are *much* faster than STWFC. They also slow down less quickly with more complex output; while STWFC takes 7.4 times as long to generate *Sokoban* as *Field*, block and diff take only 2.8 and 2.9 times as long, respectively. Interestingly, block shows similar slowdown to STWFC for *Maze*, taking 4.6 times as long as *Field* (compared to 5.25 times for STWFC). It is unclear why *Maze* is so much slower than *Sokoban* for block only.

As STWFC is based on WFC, the default strategy is to collapse tiles until solved or a contradiction is found — thus it is possible for generating a level to fail. WFC can also have difficulty with global constraints, such as only having a single player in the level. Another advantage of the new methods over STWFC is that they never fail to generate (STWFC fails up to 50% of the time) or spawn a second P (STWFC spawns a second P up to 60% of the time).

Looking at effective length and trivial solutions, for Field and Sokoban-Small, it appears that the block method is mostly similar to STWFC, which may make sense as it was designed to be a similar approach. However, in Maze, all generated filmstrips were trivial for both new methods and on average were solved in a single timestep. This could be due to the relative "denseness" of the walls in the game making these solutions much easier to find, as well as adding all patterns to pool 0 allowing a pattern with the level completed to be used in the first frame. In comparison STWFC has an average effective length of 3.5 frames and is only trivial half of the time. STWFC's increased effective length is likely a consequence of its level initialization placing the player and the door at random positions in the first timestep.

Generating Larger Levels Interactively

Method

To give a designer more control over generated filmstrips, we developed a simple interactive application to overlay constraints on tiles anywhere on the filmstrip. Screenshots are shown in Figure 1. The application displays a single window, showing the filmstrip as a grid of tiles and allows the user to scroll to view the entire filmstrip.

More specifically, to start the application, the same inputs as the generation process (e.g. configuration options and various grids) are needed and these cannot be edited at runtime. The application allows the designer to interactively overlay additional constraints in the form of tiles required to be at specific locations in the filmstrip — essentially adding another match grid that can be interactively edited. This allows the designer to constrain tiles at any point in spacetime.

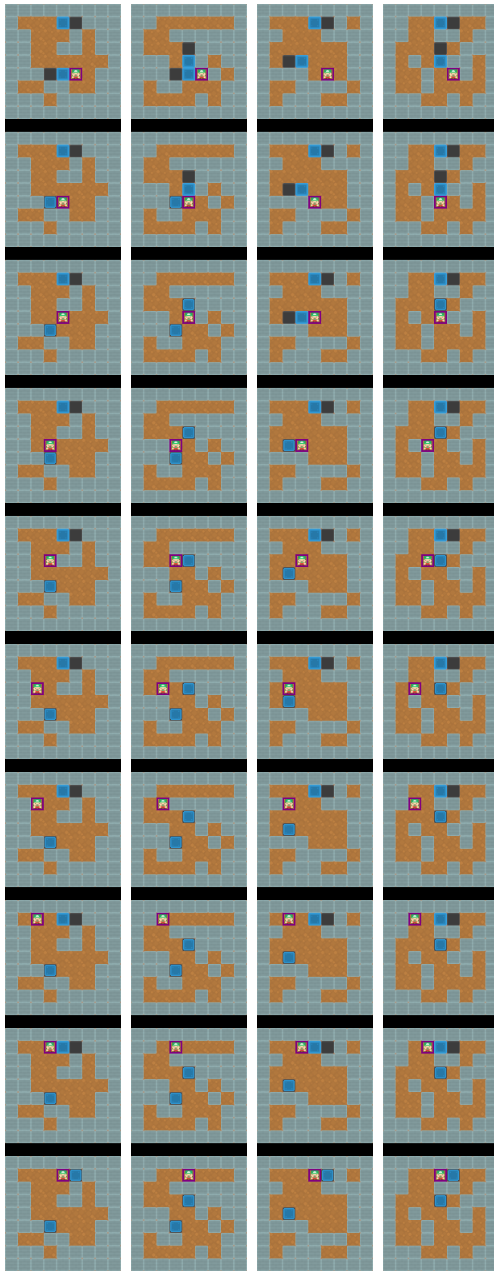

Figure 6: Sokoban filmstrips generated with the same path. Constrained tiles are outlined in purple.

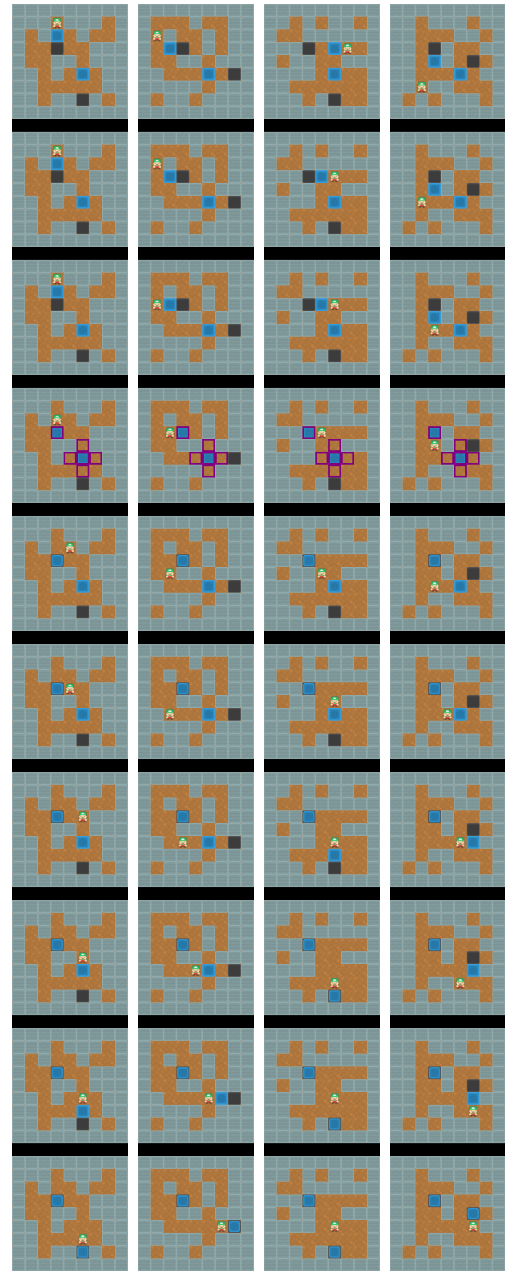

Figure 7: Sokoban filmstrips that go through the same intermediate tile arrangement. Constrained tiles are outlined in purple.

Thus they can edit not just the initial level itself, but also edit frames in the resulting generated solution filmstrip. Whenever the overlaid constraints change, the application starts to regenerate a filmstrip in the background and presents it to the user when it is ready; in the case that no solution can be found, no filmstrip is displayed. This application could be considered a form of mixed-initiative level editing: the user can specify some constraints on tiles at certain points in spacetime, and the system will attempt to find a filmstrip that satisfies them.

We note that in this approach, editing tiles in spacetime does not necessarily require them to be part of *every* solution, just that there is *some* solution that uses them. That is, players may be able to find a different solution than the one the designer constrained. Also, it is very possible for a designer to overlay constraints that make generating a filmstrip impossible.

Due to the performance improvements of the Sturgeon-based methods, here we use the diff method in two more

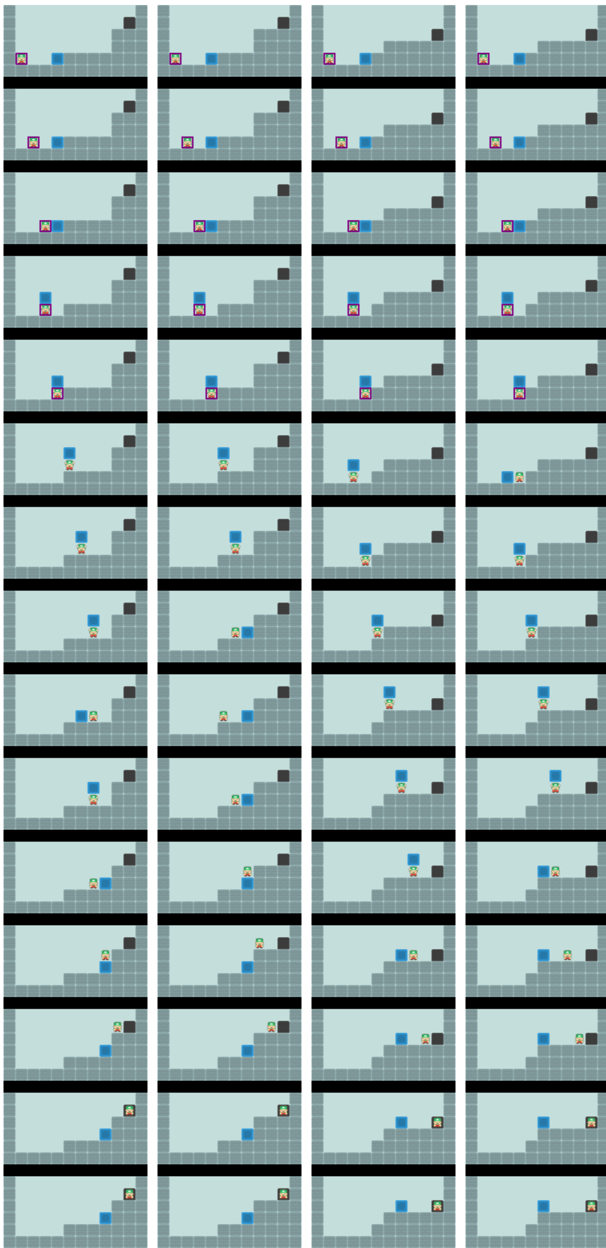

Figure 8: Block Dude filmstrips generated with the same initial path. Constrained tiles outlined in purple.

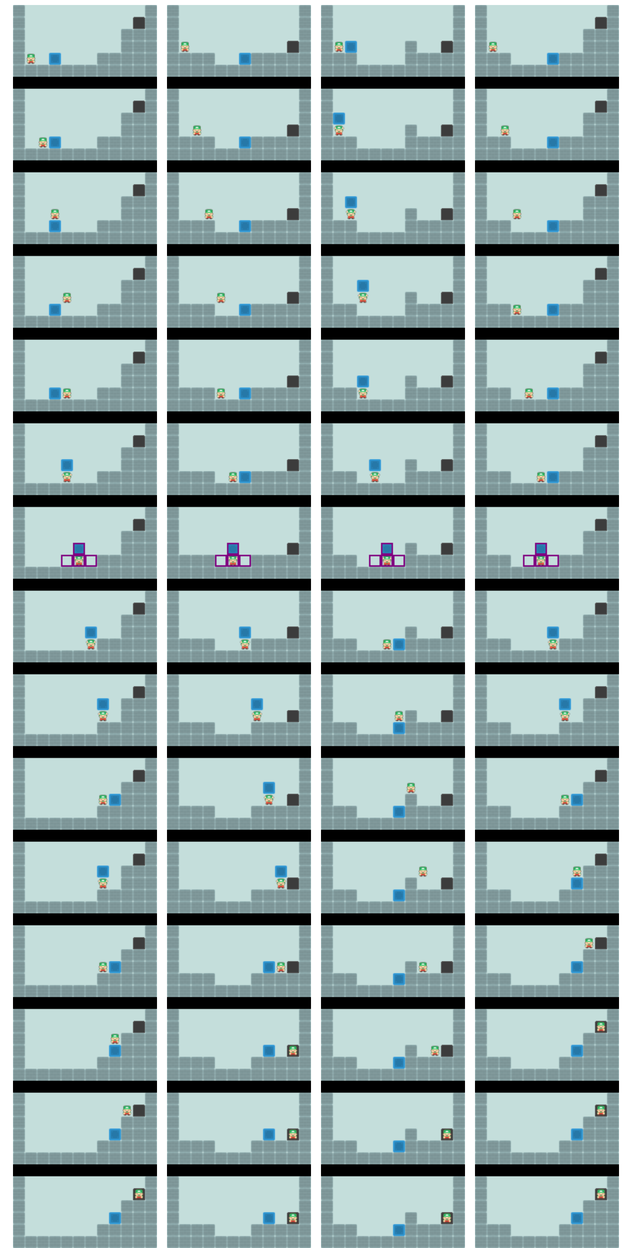

Figure 9: Block Dude filmstrips generated that go through the same intermediate tile arrangement. Constrained tiles outlined in purple.

complex games.

Sokoban-Large — Similar to Sokoban-Small, but we use a filmstrip of the first level from the standard Sokoban game (Thinking Rabbit 1928), rotated and flipped. Another notable difference is that the player walks over targets in this example. The first frame must have one player. We generate levels with effective size $7row \times 7col \times 10frame$ (plus a padding of wall around each frame) with two crates, and effective size $10row \times 10col \times 20frame$ with three crates.

Block Dude — A side-view puzzle game where the player can pick up, move, and stand on crates to reach the exit (Detached Solutions 2007). We use a filmstrip of the first level for training. As Block Dude has gravity, we only mirror it. We use a slightly simplified variation that does not track the direction the player is facing. The first frame must have one player on the left and one door on the right. We generate levels with effective size $4row \times 10col \times 15frame$ (plus padding around each frame) with one crate, and effective size $4row \times 16col \times 24frame$ with 2 crates.

Notably, for Block Dude we also added support for *context* when learning diff patterns. Context is taken into con-

sideration when the diff pattern learning method finds differing tiles between frames: context specifies additional relative tiles to add to the pattern, even if those tiles don't change. In Block Dude, the context is one tile above and below. Thus, when the player steps to the side into a blank tile, the solid tile below the player is added to the pattern, even though that tile doesn't change. This prevents the player from, for example, walking out onto empty space.

We note that, as our approach presumes that the player has control over each step, if the appropriate patterns were seen during training, generated solutions could involve moves where the player can control when crates fall due to gravity, which isn't possible in the game. However, there were no instances of crates falling independently of the player in the training filmstrip we used.

The first frame of the training filmstrips for each game are shown in Figure 5. Tiles also have sprite images associated with them for generating filmstrips, which come from Kenney (Kenney 2010). Figure 1 shows a screenshot of the application using Sokoban, with an intermediate frame having overlay constraints of a block being on a target, and a block off a target with blanks around it.

To get a high-level overview of the approach, we again generated 10 levels from each game. A summary can be found in Table 2.

While this application can allow a user to iteratively edit and refine a filmstrip, we explore a few basic possibilities for generating variations with constrained tiles.

Variations on Player Solution Path

One application of the spacetime editor application is generating level variations that allow the player to take the same path through the level, but the other tiles of the filmstrip to be rearranged in such a way that this path still solves the level.

In both games, we explored this by generating a filmstrip in the application, copying the complete or partial path of the player into the overlay constraints; and then generating new filmstrips with those overlay constraints.

In Sokoban-Large, we copied the entire player path, and generated filmstrips are shown in Figure 6. Visually, the variations place the crates and targets at various places throughout the level, in such a way that a player moving through the level in the same way ends up pushing the crates onto the targets. However, the crates all start next to the targets. It is possible that additional constraints on the first frame could rule this out.

In Block Dude, we copied only the first few frames of the path, in which the player moves forward and picks up the crate. Generated filmstrips are shown in Figure 8. Visually, the filmstrips are fairly similar, and in fact the crate is not actually required in all of them.

Variations on Intermediate Tile Arrangement

Another application of the spacetime editor application is overlaying an arrangement of tiles in a certain frame that some solution to the level must go through. Again, it may be possible that there are solutions that don't go through this arrangement, but there must at least be one.

In both games, we explored this by adding overlay constraints on a desired frame and then generating new filmstrips with those overlay constraints.

In Sokoban-Large, we set a tile arrangement with a crate on a target and a crate not on a target surrounded with blanks; generated filmstrips are shown in Figure 7. Visually, again the variations place the crates, targets, and player at various places throughout the level. Here, the crates are not always placed next to targets.

In Block Dude, we set the player to be carrying the block next to blanks; generated filmstrips are shown in Figure 9. Again the filmstrips are fairly similar; the crate is not required in all of them, although the player does pick it up.

Conclusion

In this work we presented a constraint-based approach to learning and applying spacetime constraints for 2D tile- and turn-based puzzle games. We described two methods, the block and diff methods, that work in a filmstrip representation of a level's solution, which unrolls the time dimension. We compared to our previous implementation STWFC, and developed and described an interactive spacetime filmstrip editor along with a few applications.

We found that the levels generated are often simple, e.g. placing crates next to their associated targets, or not requiring the player to pick up a crate. While we expect that such filmstrips can be ruled out by more constraints, we are interested in a user study with designers to explore how they would like to use the application, if they find the ability to add spacetime constraints at various frames in the filmstrip useful, and if it scales to puzzles they would like to create.

Since the filmstrip representation encodes 3D spacetime into a 2D grid, it may be useful to explore how this encoding works when other existing 2D PCGML techniques are applied, such as GANs (Volz et al. 2018) or diffusion models (Dai et al. 2024). It may also be interesting to combine the learned patterns from multiple games to create new games that blend different mechanics (Sarkar and Cooper 2021).

Another interesting area is automatically determining spacetime pattern shapes. In STWFC and the block method, the block pattern shape must be provided in advance. To address the presence of gravity in Block Dude, we added support for context to the diff method that also must be provided. If the pattern shapes are incorrect, undesirable things may occur such as the player being able to walk on air. Approaches to automatically (or semi-automatically) determining the diff context, potentially by detecting such errors in the generated levels, could help improve the overall system. It would also be useful to learn pattern "priorities", such as that in Block Dude, gravity needs to take effect before the player can move.

Acknowledgments

The authors would like to thank Emily Halina for helpful discussions. The authors acknowledge Writefull for spelling and grammar improvements.

References

Alvarez, A.; Dahlskog, S.; Font, J.; Holmberg, J.; Nolasco, C.; and Österman, A. 2018. Fostering creativity in the mixed-initiative evolutionary dungeon designer. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, FDG '18, 1–8. New York, NY, USA: Association for Computing Machinery. ISBN 978-1-4503-6571-0.

Audemard, G.; and Simon, L. 2009. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI'09, 399–404. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Banerjee, R.; Yip, J.; Lee, K. J.; and Popović, Z. 2016. Empowering children to rapidly author games and animations without writing code. In *Proceedings of the The 15th International Conference on Interaction Design and Children*, IDC '16, 230–237. New York, NY, USA: Association for Computing Machinery. ISBN 978-1-4503-4313-8.

Bruce, J.; Dennis, M.; Edwards, A.; Parker-Holder, J.; Shi, Y.; Hughes, E.; Lai, M.; Mavalankar, A.; Steigerwald, R.; Apps, C.; Aytar, Y.; Bechtle, S.; Behbahani, F.; Chan, S.; Heess, N.; Gonzalez, L.; Osindero, S.; Ozair, S.; Reed, S.; Zhang, J.; Zolna, K.; Clune, J.; Freitas, N. d.; Singh, S.; and Rocktäschel, T. 2024. Genie: generative interactive environments. ArXiv:2402.15391 [cs].

Cooper, S. 2022a. Constraint-based 2D tile game blending in the Sturgeon system. In *Proceedings of the Experimental AI in Games Workshop*.

Cooper, S. 2022b. 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): 26–36.

Cooper, S. 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*, FDG '23, 1–9. New York, NY, USA: Association for Computing Machinery. ISBN 978-1-4503-9855-8.

Cooper, S. 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): 254–255.

Cooper, S.; and Sarkar, A. 2020. Pathfinding agents for platformer level repair. In *Proceedings of the Experimental AI in Games Workshop*, 5.

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*, AcademicMindTrek '14, 200–206. Tampere, Finland: Association for Computing Machinery. ISBN 978-1-4503-3006-0.

Dai, S.; Zhu, X.; Li, N.; Dai, T.; and Wang, Z. 2024. Procedural level generation with diffusion models from a single example. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(9): 10021–10029. Number: 9.

Detached Solutions. 2007. PuzzPack v2.0.

Eén, N.; and Sörensson, N. 2003. An extensible SAT-solver. In Giunchiglia, E.; and Tacchella, A., eds., *Theory and Applications of Satisfiability Testing*, number 2919 in Lecture Notes in Computer Science, 502–518. Springer Berlin Heidelberg. ISBN 978-3-540-20851-8 978-3-540-24605-3.

Facey, K.; and Cooper, S. 2024. Toward space-time WaveFunctionCollapse for level and solution generation. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 20(1): 25–34. Number: 1.

Furnas, G. W. 1991. New graphical reasoning models for understanding graphical interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 71–78.

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

Gumin, M. 2022. MarkovJunior. https://github.com/mxgmn/MarkovJunior/.

Guzdial, M.; Li, B.; and Riedl, M. O. 2017. Game engine learning from video. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, 3707–3713. Melbourne, Australia: AAAI Press. ISBN 978-0-9992411-0-3.

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*, CHI '19, 1–13. New York, NY, USA: Association for Computing Machinery. ISBN 978-1-4503-5970-2.

Ha, D.; and Schmidhuber, J. 2018. World models. ArXiv:1803.10122 [cs].

Ignatiev, A.; Morgado, A.; and Marques-Silva, J. 2018. PySAT: a Python toolkit for prototyping with SAT oracles. In *Theory and Applications of Satisfiability Testing – SAT 2018*, 428–437.

Jain, R.; Isaksen, A.; Holmgård, C.; and Togelius, J. 2016. Autoencoders for level generation, repair, and recognition. In *Proceedings of the ICCC workshop on computational creativity and games*, volume 9.

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*, 1–10.

Karth, I.; and Smith, A. M. 2019. Addressing the fundamental tension of PCGML with discriminative learning. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, FDG '19, 1–9. New York, NY, USA: Association for Computing Machinery. ISBN 978-1-4503-7217-6.

Kenney. 2010. Home. https://www.kenney.nl/.

Kim, H.; Hahn, T.; Kim, S.; and Kang, S. 2020a. Graph based wave function collapse algorithm for procedural content generation in games. *IEICE Transactions on Information and Systems*, E103.D(8): 1901–1910.

Kim, S. W.; Zhou, Y.; Philion, J.; Torralba, A.; and Fidler, S. 2020b. Learning to simulate dynamic environments with GameGAN. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Langendam, T. S. L.; and Bidarra, R. 2022. miWFC-Designer empowerment through mixed-initiative Wave Function Collapse. In *Proceedings of the 17th International Conference on the Foundations of Digital Games*, 1–8.

Liapis, A.; Yannakakis, G. N.; and Togelius, J. 2013. Sentient Sketchbook: Computer-Aided Game Level Authoring. In *Proceedings of the 8th Conference on the Foundations of Digital Games*, 213–220.

Liffiton, M. H.; and Maglalang, J. C. 2012. A cardinality solver: more expressive constraints for free. In *Theory and Applications of Satisfiability Testing – SAT 2012*, 485–486.

Liu, C. K.; Hertzmann, A.; and Popović, Z. 2005. Learning physics-based motion style with nonlinear inverse optimization. *ACM Trans. Graph.*, 24(3): 1071–1081.

Sarkar, A.; and Cooper, S. 2021. Dungeon and platformer level blending and generation using conditional VAEs. In *2021 IEEE Conference on Games (CoG)*, 1–8. ISSN: 2325-4289.

Shaker, N.; Togelius, J.; and Nelson, M. J. 2016. *Procedural Content Generation in Games*. Springer International Publishing.

Shu, T.; Wang, Z.; Liu, J.; and Yao, X. 2020. A novel CNet-assisted evolutionary level repairer and its applications to Super Mario Bros. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, 1–10.

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

Snodgrass, S.; and Ontanon, S. 2015. A hierarchical MdMC approach to 2d video game map generation. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 11(1): 205–211. Number: 1.

Stålberg, O. 2021. Townscaper. https://www.townscapergame.com/. Accessed: 2025-06-25.

Summerville, A.; and Mateas, M. 2016. Super mario as a string: Platformer level generation via lstms. *arXiv preprint arXiv:1603.00930*.

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. *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment; Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*.

Sumner, M.; Saini, V.; and Guzdial, M. 2024. Mechanic maker: accessible game development via symbolic learning program synthesis. In *Proceedings of the Twentieth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 20 of *AIIDE '24*, 235–244. Lexington: AAAI Press. ISBN 978-1-57735-895-4.

Thinking Rabbit. 1928. Sokoban. Game.

Togelius, J.; Yannakakis, G. N.; Stanley, K. O.; and Browne, C. 2011. Search-based procedural content generation: a taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3): 172–186.

Viana, B. M. F.; Pereira, L. T.; Toledo, C. F. M.; dos Santos, S. R.; and Maia, S. M. D. M. 2022. Feasible–infeasible two-population genetic algorithm to evolve dungeon levels with dependencies in barrier mechanics. *Applied Soft Computing*, 119: 108586.

Volz, V.; Schrum, J.; Liu, J.; Lucas, S. M.; Smith, A.; and Risi, S. 2018. Evolving Mario levels in the latent space of a deep convolutional generative adversarial network. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 221–228. ACM.

Withington, O. 2020. Illuminating Super Mario Bros: quality-diversity within platformer level generation. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, 223–224.

Witkin, A.; and Kass, M. 1988. Spacetime constraints. *ACM SIGGRAPH Computer Graphics*, 22(4): 159–168.

Yannakakis, G. N.; Liapis, A.; and Alexopoulos, C. 2014. Mixed-initiative co-creativity. In *Proceedings of the 9th International Conference on the Foundations of Digital Games*.

Yu, J.; Qin, Y.; Wang, X.; Wan, P.; Zhang, D.; and Liu, X. 2025. GameFactory: creating new games with generative interactive videos. ArXiv:2501.08325 [cs].

Zhang, H.; Fontaine, M.; Hoover, A.; Togelius, J.; Dilkina, B.; and Nikolaidis, S. 2020. Video game level repair via mixed integer linear programming. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 16(1): 151–158.