Toward Space-Time WaveFunctionCollapse for Level and Solution Generation

Kaylah Facey, Seth Cooper

Khoury College of Computer Sciences, Northeastern University
facey.k@northeastern.edu, se.cooper@northeastern.edu

Abstract

WaveFunctionCollapse (WFC) is a constraint-satisfaction-based approach to procedural content generation via machine learning (PCGML). It is relatively straightforward to implement and requires very little example data, making it a popular approach. Generated game levels are guaranteed to look locally similar to their example tilemaps; however, local adjacency rules often fail to capture global solvability rules, potentially making many such levels unplayable. Existing approaches to improving the solvability of WFC-generated levels typically require adding additional game-specific information in the form of global constraints, substantially increasing the complexity and time required for setup. The purpose of this work is to explore whether using level solutions as example data can allow WFC to learn solvability constraints and game mechanics. We have implemented a novel space-time approach that uses three-dimensional space-time blocks representing solutions to 2D levels as both input and output. Experiments using this method show that space-time WFC is capable of demonstrating localized game mechanics and creating small playable levels with given solutions. However, levels are slow to generate, and some high-level constraints are still not captured.

Introduction

The purpose of procedural content generation (PCG) for game levels is to allow game designers to create new levels with minimal effort by automating all or part of the process. Many existing approaches, however, require designers to input a huge amount of game-specific information in the form of either example data or extensive rules authoring, reducing the cost and time saving. WaveFunctionCollapse (WFC) (Gumin 2016) was introduced as a machine learning (ML) approach to generate images and level maps from minimal example data (as little as one example level), and its ease of implementation and generalizability have made it popular for PCGML (Summerville et al. 2018). However, though images and levels generated from WFC generally look good, because constraints are generated from local regions of the input, they often fail to capture high-level constraints (e.g. Cheng, Han, and Fei (2020)). Current approaches to improving the solvability of procedurally generated levels involve annotating 2D example levels with solution information (e.g. Summerville and Mateas (2016); Lee, Partlan, and Cooper (2020)) or incorporating high-level constraints given by designers (e.g. Cooper (2023b)). Annotated paths are easy to generate using playtraces but treat a solution as purely spatial, losing information from the temporal dimension and failing to capture how multiple solution actions may occur at the same spatial location at different times (Sorochan et al. 2021). High-level constraints, on the other hand, require much of the same game-specific rules authoring that WFC originally avoided.

In this work, we attempt to capture local game mechanics and solution constraints using a novel *space-time* method that includes both spatial and temporal dimensions of example level solutions as input to WaveFunctionCollapse (here referred to as space-time WaveFunctionCollapse, or STWFC). This method requires no more work to implement than three-dimensional WFC, and using it for new games requires minimal game-specific information other than example level solutions that can be generated using playtraces.

Our experiments show that STWFC is capable of generating levels with valid solutions demonstrating game mechanics and ending conditions, though the method is limited by being slow to execute and requiring that each timestep in a solution affects only a small region in space.

Related Work

WaveFunctionCollapse

WaveFunctionCollapse (WFC) is an algorithm originally introduced by Gumin (2016) for use in generating tilemaps and images that are locally similar to an example grid. Input grids are divided into overlapping $N \times M$ regions that represent all valid output patterns, and a new grid is generated by iteratively "collapsing" the cell with the fewest remaining valid patterns by selecting one of them, and then propagating the result of that collapse. If no valid pattern exists for a cell, the algorithm fails. In failure cases typically the algorithm must be re-run, though it is common to implement backtracking (Karth and Smith 2022, 2017; Kim, Seo, and Kang 2024).

WFC has gained prominence in PCG because it is relatively simple to understand, easy to implement, and it requires relatively little example data. Since its introduc-

tion, WFC has been extended to work with high-level constraints (Sandhu, Chen, and McCoy 2019; Cheng, Han, and Fei 2020), graph structures instead of rectangular grids (Kim et al. 2019; Cooper 2023a), and hierarchical patterns (Beukman et al. 2023; Alaka and Bidarra 2023; Nie et al. 2024). WFC has also been used to solve logic puzzles (Kim et al. 2019; Kim, Seo, and Kang 2024). An unpublished project to create cellular automata, documented by a handful of tweets (Rix 2017e,b,c,a,d, 2018), provides the only public evidence (to our knowledge) of prior use of WFC with a time dimension; it is cited by Gumin (2016) among examples of WFC. The use of WFC for procedural content generation was recently surveyed by Karth and Smith (2022).

Generating Complete Game Levels

A lot of work has been done to improve or guarantee the solvability of procedurally generated game levels.

One way to improve solvability has been to include level solution data. One common method is to add path annotations to 2D levels, which has been found to improve the likelihood that generated levels are completable (Summerville and Mateas 2016; Lee, Partlan, and Cooper 2020). However, Sorochan et al. (2021) find that 2D annotations can be insufficient to capture all the dependencies of a path through a level. It's also common to enforce the existence of a solution path by incorporating an A* agent into level generation (Babin and Katchabaw 2021; Cooper and Sarkar 2020; Kartal, Sohre, and Guy 2016; Snodgrass and Ontanón 2017; Sarkar and Cooper 2020). Snodgrass and Ontanón (2017) also generates levels by sampling training data with the aide of a multi-dimensional Markov chain representing the likelihood of player actions given their surroundings and previous actions.

Other approaches incorporate solvability in the form of designer-created game-specific constraints (Horswill and Foged 2021; Snodgrass and Ontanón 2016; Smith, Whitehead, and Mateas 2011; Smith et al. 2012) or grammars (Font et al. 2016; Smith et al. 2009). *MarkovJunior* (Gumin 2022) introduces space-time considerations by defining a set of "rewrite rules" specifying ways a grid can change over time. Cooper (2023b) uses rewrite rules for level generation by incorporating them into *Sturgeon* (Cooper 2022), which already used 2D constraints. Our work was partially inspired by the idea of learning rewrite rules by example rather than writing them manually.

System Overview

STWFC is, at its core, 3D WFC with one dimension representing time, and the two other dimensions representing a 2D space. We call these 3D grids *space-time blocks*. Similar to standard WFC, STWFC learns patterns from examples. Whereas in standard WFC the examples are 2D levels, in STWFC the examples are space-time blocks representing both a level *and* the steps to solve it, progressing along the time dimension. While standard WFC learns local $N \times M$ patterns from the examples, STWFC learns local $T \times Y \times X$ patterns (where T is the time dimension). To generate a new level, these space-time patterns are then assembled into a new space-time block, where the first timestep is the 2D level, and each step to solve the generated level is appended along the time dimension. If STWFC succeeds, the output is a new 2D level with its solution; if it fails, an empty 3D grid is returned, and the algorithm may be rerun. As one dimension represents time, STWFC also includes some special handling of boundary conditions along the edges of the blocks and what transformations (e.g. rotations) can be applied to examples.

Core WaveFunctionCollapse

Our implementation of STWFC is at its core an open-source Python implementation (Le 2019) of WaveFunctionCollapse (WFC) that supports three dimensions, which we forked [1] to add the configurations we required. The original implementation is unchanged except where explicitly noted.

The inputs and outputs of STWFC are 3D grids of dimensions time (T), height (Y), and width (X). WFC is composed of two phases: pattern finding and generation. In the pattern finding phase, valid patterns for the output are discovered by passing an overlapping $t \times y \times x$ window (where $t \leq T$, $y \leq Y$, and $x \leq X$) over the input. The shape of the window is configured by the user and should be no smaller than the smallest region in space-time affected by any game mechanic. For example, if there is a jump mechanic that allows a player to move up two spaces over the course of two timesteps, the window should be no smaller than $3T \times 1X \times 3Y$. Sometimes, as discussed below in **Experiments: Pattern Shape**, the pattern window must be made larger to capture mechanics properly.

In the generation phase, a new space-time block is initialized and filled iteratively. Typically, a WFC grid starts completely blank; STWFC uses a custom initialization function discussed below in **Input**. Each cell of the grid contains a list of potential patterns for that cell, and the algorithm terminates when either every cell has "collapsed" to exactly one pattern (success) or any cell has no remaining valid patterns (failure). In the failure case, the grid is returned empty, and the process may be restarted. To determine the order in which to collapse cells, WFC uses the greedy heuristic of finding the cell with the lowest entropy and choosing a random value for that cell. Then the information is propagated recursively throughout the grid.

Input and Output

The inputs and outputs of STWFC are example level *solutions*. Solutions are represented as a series of timesteps that are are stacked on top of each other, forming a three-dimensional space-time block with dimensions time (T), height (Y), and width (X). Input solutions may be created by hand or generated using human or agent playtrace data. Additionally, a pattern shape must be provided for finding patterns in the input.

As has been done in previous work (e.g. Cheng, Han, and Fei (2020)), we can apply transformations to the input to reduce the amount of example data required. The possible

[1] https://github.com/flaneuseh/wave-function-collapse/releases/tag/AIIDE

transformations are reversing the direction of any axis and rotating the block by 90, 180, or 270 degrees in any plane. The open-source WFC we used (Le 2019) by default applies almost all possible transformations; we amend this to be configurable by the user. Transformations may be applied to the solution object as a whole or to individual patterns discovered during the pattern finding phase of WFC (discussed above in **Core WaveFunctionCollapse**). Transformations are valid if all of the game's mechanics remain valid after the transformation. For example, for most top-down games in which all movement can occur in all four directions, both the X and Y axes may be reversed, and the XY plane may be rotated by 90, 180, and 270 degrees. For a sideview game, on the other hand, typically the only transformation that may be applied is to reverse the X axis. Rarely, if all moves in a game may be done in reverse, it may make sense to reverse the T axis; however, it does not make sense to rotate the TX or TY planes.

After applying input transformations, the input to STWFC may be padded in any direction. To enforce that generated solutions reach a solved state (and therefore that the level is solvable), all example solutions are padded in the "future" direction of the T axis with a special character (T) indicating that the level is solved. In the pattern finding step, only patterns used in valid solutions will include Ts. The output grid is initialized with the same padding, so WFC is forced to only include valid solution patterns in the final timesteps. Generally, solutions must be padded in the X and Y axes as well as in the T axis. This is because WFC treats the region surrounding the grid as uninitialized space. Moving game objects can then "walk" into and out of the surrounding region at any time, seeming to "appear" and "disappear" from the level.

Finally, STWFC requires the specification of an initialization function that describes the output grid before padding is applied or initial cell pattern lists are determined. The purpose of the initialization function is to ensure that WFC does not generate a solution without a valid solved state, such as when all tiles are static objects. The initialization function should at minimum place some dynamic object with a clearly defined end state that will be captured during pattern finding. For example, if the aim of the game is for a player to reach some goal, either the player or the goal should be placed. (In this scenario, it suffices to place *either* the player or the goal as the existence of one implies the other). The use of an initialization function to constrain WFC output has been done in previous work (e.g. (Cheng, Han, and Fei 2020)).

Experiments

To test our approach, we ran experiments using several example games, aiming to guarantee the following for each generated level solution:

G1 Every change to the level over the course of the solution represents a valid gameplay mechanic.

G2 The solution reaches a solved state (e.g. the player reaches the goal).

To simplify our example data and configuration, we chose to use games that have the following properties:

P1 The game is 2D with a top-down view, and all mechanics may be reversed in the X or Y axis and rotated by 90, 180, or 270 degrees in the XY plane.

P2 All steps in a level solution are valid starting points for a level.

P3 Every game mechanic may be captured with a $2T \times 3Y \times 3X$ region in space-time.

Configuration

All games tested use the same input transformations and pattern shape, with almost identical padding. Game-specific details are discussed below in **Example Games**.

Input Transformations Because of **P1**, we are able to apply all available XY input transformations, simplifying our example data. Input solutions are reversed in the X or Y axis and rotated by 90, 180, or 270 degrees in the XY plane. The same input transformations are also applied on each pattern discovered during pattern finding.

Padding As discussed above, all inputs and outputs are padded in the "future" direction of the T axis with a special character (T) indicating that the level is solved. Also as discussed, the X and Y axes are padded in both directions with a character specified per-game. **P2** allows us *not* to pad the T axis in the "past" direction, simplifying our example data.

As well as STWFC's padding configuration, each input solution is given multiple copies of its final state at the end of the solution (see Figure 2). This permits output levels to reach a solution and terminate early, as patterns show a pause in the solved state.

Pattern Shape To support **G1**, we use the pattern shape of $2T \times 3Y \times 3X$ guaranteed by **P3**. Note that for two of the example games discussed below (**Example Games**: **Field** and **Maze**), the player (P) is the only object that moves, and it may move only one tile per timestep. From this it may seem as though a pattern shape of $2T \times 2Y \times 2X$ would suffice. (In the third game, **Sokoban**, the P may push a box by one tile, necessitating the $3Y \times 3X$ pattern regardless.) However, each time P moves, it also changes pattern windows such that some patterns have a P in the first timestep but not the second, or vice versa (seeming to "appear" in or "disappear" from the pattern. Because P can move in any direction, there are patterns for P appearing or disappearing in every cell in a $2Y \times 2X$ region. Padding the X and Y pattern dimensions prevents this, as if P moves into or out of the center of a 3×3 region, it must be present in both timesteps of the pattern. It does not matter when P enters or leaves a pattern on the edge; the pattern window overlaps, so surrounding patterns capture its movement (see Figure 1).

Example Games

We tested our approach with three different games, each of which was used independently.

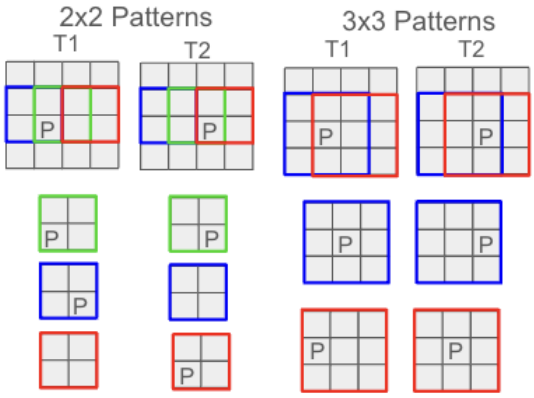

Figure 1: Objects can appear and disappear from any cell in a 2×2 pattern but can never appear or disappear in the center cell of a 3×3 pattern. 1. P can disappear or appear from any cell in a 2×2 pattern (imagine the first row if P moved left instead of right). 2. With a 3×3 pattern, P remains in the pattern for both timesteps when it moves into or out of the center cell.

Field **Field** was made as a proof-of-concept game and is simply an open field in which a player P navigates to a door D. Our aim was to see if with minimal example data, STWFC could find a valid solution for an arbitrarily located P and D. For our analysis, we used two example solutions, *Field 1* (Figure 2) [2] and *Field 2* (similar but larger; see **Appendix**). To test the effect of adding or removing example data, we also ran STWFC for each solution independently.

For each solution and the set of two solutions, we ran STWFC 10 times each for grid sizes $10T \times 6Y \times 6X$ and $6T \times 4Y \times 4X$. The XY padding character we used was '_', and the initialization function for **Field** places a P and a D in random XY positions in the first timestep of the solution (Note that it is possible for P and D to be randomly assigned the same XY; in that case only the D is initialized).

Maze The next game we tested is an extension of **Field** in which walls W block the player P's movement toward the door D. With **Maze** we again wanted to see if STWFC could produce a valid solution for an arbitrarily located P and D. We used an XY padding tile of W and the same initialization function as **Field**. For all generations, we input two example solutions, *Maze 1* (see **Appendix**) and *Maze 2* (similar but larger, shown in Figure 3). We ran STWFC 10 times each for grid sizes $10T \times 6Y \times 6X$, $10T \times 4Y \times 4X$, and $6T \times 4Y \times 4X$. The $10T \times 4Y \times 4X$ size was added after $10T \times 6Y \times 6X$ grids took an average of over 3 hours to generate.

Sokoban Our final test game is an implementation of the logic puzzle *Sokoban*, in which the player P must push one or more blocks B such that every block lands on a target O (a block on a target is represented by G). With **Sokoban**, we aimed to test how STWFC would handle a more complex mechanic. We used an XY padding tile of '_', and all generations used the same two example solutions, one with wall W barriers (*Sokoban 2*; partially given in Figure 5) and one without them (*Sokoban 1*; Figure 4). **Sokoban** output solutions were initialized with a randomly placed B, as the location of the Bs is more important to the solution than the location of the P. We ran STWFC 10 times each for grid sizes $6T \times 4X \times 4Y$ and $7T \times 4X \times 4Y$. Generations for larger sizes were terminated after STWFC ran for several hours without producing a result.

[2] Level images generated with https://github.com/crowdgames/level2image

Figure 2: *Field 1* example level (note the repeated solution state).

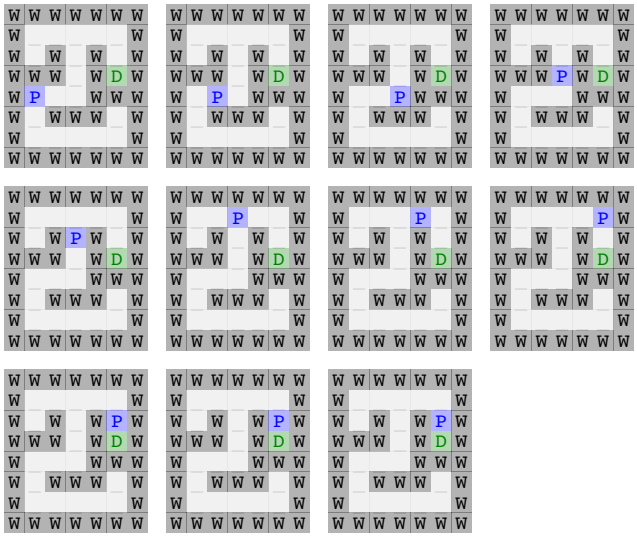

Figure 3: *Maze 2* example level.

Results

All code, configuration, and results used for these results are available on Open Science Framework [3]. For each game con-

[3] https://osf.io/ag8mz/

Figure 4: *Sokoban 1* example level.

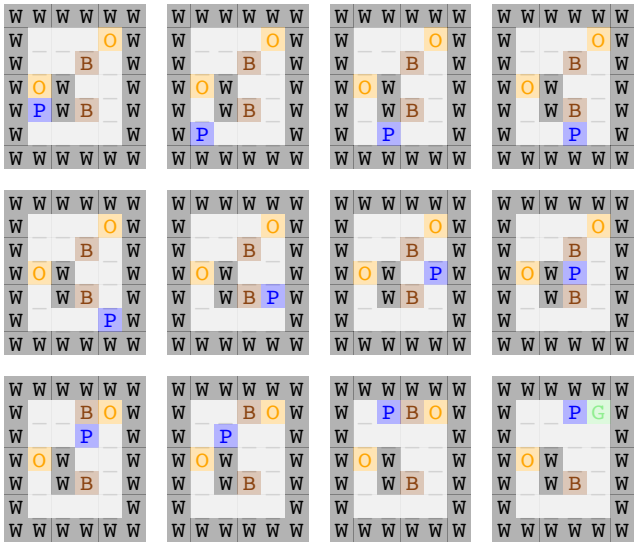

Figure 5: *Sokoban 2* example level (first 12 timesteps). Entire level in **Appendix**

figuration, we took the following measures of success over the 10 STWFC runs:

Percent Success For what percentage of runs does STWFC generate a complete output solution, rather than failing?

Average Time What is the average time STWFC takes to either arrive at a solution or fail?

Average Effective Length For finished solutions, how many timesteps does the solution take on average to reach a stable state that does not change again?

Percent Trivial When a solution is found to a level, the rest of the WFC problem is reduced to filling in the static area of the level and copying the solution state for the rest of the available timesteps. Therefore we consider generated solutions where a stable state is reached in 2 or fewer timesteps from the initial state to be "trivial".

We also noticed that some levels included an extra P object (with as many as 4 Ps total), breaking the expectation that there should be at most one player. Extra Ps occur because our method does not support global constraints. Because the initialized level is mostly nil, including at the first timestep, the generator is able to add multiple Ps, which navigate towards a goal state simultaneously (see Figure 6). Ps must be kept sufficiently separated so that they do not enter the same pattern window and cause a local constraint violation; therefore, extra Ps are more common in grids with a larger XY area. This might be prevented, for example, by initializing levels with a single P on the first timestep and removing all patterns containing a P from all other cells in the first timestep.

We tracked how many levels contain an extra P. We include measurements of average effective length and percent of trivial effective lengths for levels with and without extra Ps.

Field

The results for **Field** are given in Table 1. We find that the larger grids are much slower to generate than the smaller grids. Larger grids are 3.75 times the size of smaller grids (360 vs 96 cells), but they take at least 10 times as long to generate. Increasing the amount of example data, on the other hand, appears to proportionally increase the amount of time taken; *Field 2* is about 2.4 times larger than *Field 1* and similarly takes around twice as long ($19s$ vs $10s$) to generate, and using both example solutions together takes a little under the same amount of time as adding the independent times together ($27s$ vs $29s$). Larger grids are also less likely to generate successfully except for the case where only LV1 (Figure 2) is used. Finally, larger grids are more likely to contain an extra player P.

We looked more closely at the relationship between levels that have trivial solutions and levels that contain extra player Ps, hypothesizing that adding a P is likely to make the solution more trivial, as the added P can be placed closer to the door D added by the initialization function (an extra D can then be added close to the original P, so that both Ps reach the ending condition). We found, however, that levels with multiple Ps are roughly equally as likely as levels with one P (52% vs 50%) to have a trivial solution. Further, when we took into account levels that were "trivially initialized" with a P and a D within two cells of each other, we found that adding a second P is almost equally likely to give a trivially initialized level a non-trivial solution as it is to give a non-trivially initialized level a trivial solution (58% vs 64%).

A qualitative look at the generated **Fields** reveals that they successfully demonstrate the movement mechanic that a player P can move into any orthogonally adjacent blank space '_' and the ending condition that a P must be orthogonally adjacent to a door D (see Figure 7). We noticed that in some levels, a P stops moving for one or more timesteps.

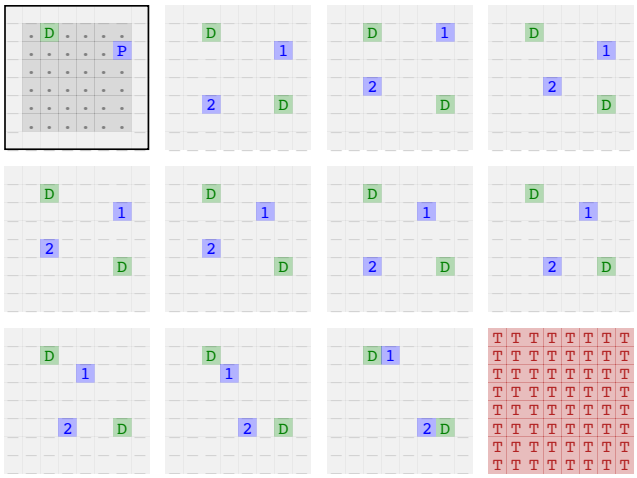

Figure 6: Generated $10T \times 6Y \times 6X$ **Field** demonstrating 2 Ps (manually labelled here as 1 and 2) moving simultaneously. The top left is the first timestep of the solution before grid population.

Grid Size	Success	Time	Extra P
$6T \times 4Y \times 4X$	0.90	$0m27s$	0.44
(LV1 only)	0.50	$0m10s$	0.60
(LV2 only)	0.80	$0m19s$	0.25
$10T \times 6Y \times 6X$	0.50	$6m16s$	1.00
(LV1 only)	0.50	$1m57s$	0.80
(LV2 only)	0.50	$3m27s$	1.00
Grid Size	Eff. Length(*)		Trivial(*)
$6T \times 4Y \times 4X$	3.9(3.6)		0.44(0.60)
(LV1 only)	3.2(5.0)		0.60(0.00)
(LV2 only)	3.1(3.8)		0.75(0.67)
$10T \times 6Y \times 6X$	9.8(-)		0.00(-)
(LV1 only)	4.8(8.0)		0.60(0.00)
(LV2 only)	7.2(-)		0.40(-)

Table 1: Results for **Field**. (*) denotes results with levels containing extra Ps removed.

On investigation, we realized that *Field 2*'s first timestep was unintentionally duplicated when creating the example solution. This does not break the resulting solutions but illustrates the effect of small changes in example data.

Maze

The results for **Maze** are given in Table 2. Compared to **Fields** (using both example solutions), **Mazes** were less likely to return a completed level (60% vs 90%) and took many times longer to generate, despite the **Maze** example data being less than half the size (916 vs 2640 cells). The generation time of **Mazes** also increases more steeply with the size of the output grid; the largest output grid takes 27 times longer to generate than the smallest, whereas for **Field**, the larger only took 14 times as long.

Mazes are also much less likely to generate an extra player P; while **Fields** had an extra P 60% of the time for small grids and (100%) of the time for large grids, **Mazes** only generated an extra P in the largest size grid and then only 60% of the time. In addition to extra Ps, three of the largest size **Mazes** included completely inaccessible regions of blank space (see Figure 9), which the example **Mazes** are designed to avoid.

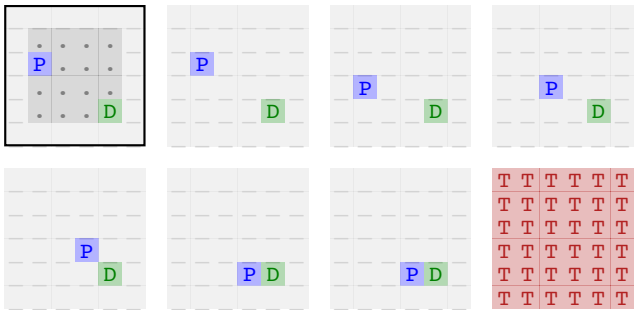

Figure 7: Generated $6T \times 4Y \times 4X$ **Field** demonstrating the movement mechanic and ending condition. The top left is the first timestep of the solution before grid population.

Grid Size	Success	Time	Extra P
$6T \times 4Y \times 4X$	0.60	$0h7m21s$	0.00
$10T \times 4Y \times 4X$	0.30	$1h16m23s$	0.00
$10T \times 6Y \times 6X$	0.50	$3h18m59s$	0.60
Grid Size	Eff. Length(*)		Trivial(*)
$6T \times 4Y \times 4X$	3.5(3.5)		0.50(0.50)
$10T \times 4Y \times 4X$	4.0(4.0)		0.67(0.67)
$10T \times 6Y \times 6X$	3.2(5.5)		0.60(0.00)

Table 2: Results for **Maze**. (*) denotes results with levels containing extra Ps removed.

A qualitative look at the generated **Mazes** shows that they successfully demonstrate the maze movement mechanic that a player P can move into any orthogonally adjacent blank space '_' while avoiding wall Ws (see Figure 8), in addition to the same ending condition as **Field** (P must be orthogonally adjacent to a door D).

Sokoban

The results for **Sokoban** are given in Table 3. **Sokoban** was equally as likely to return a completed solution as **Maze** (60%) and took the longest by far of all three games to generate. Though its example data is only 60% the size of **Field's** (using all example data) (1578 vs 2640 cells), **Sokoban's** smallest levels took over 100 times longer on average to generate than did the same size **Fields** ($51m10s$ vs $27s$). At the same size, **Sokoban** levels are also substantially more likely to have extra Ps (70% vs 41%) and trivial solutions (67% vs 44%).

A qualitative look at the generated **Sokobans** shows that they successfully demonstrate the mechanic that a player P can push a block B onto any blank space '_' or target O. Non-trivial solutions also demonstrate the high-level block organizing mechanic of Ps moving around Bs in order to push them in different directions (see Figure 10). **Sokobans**

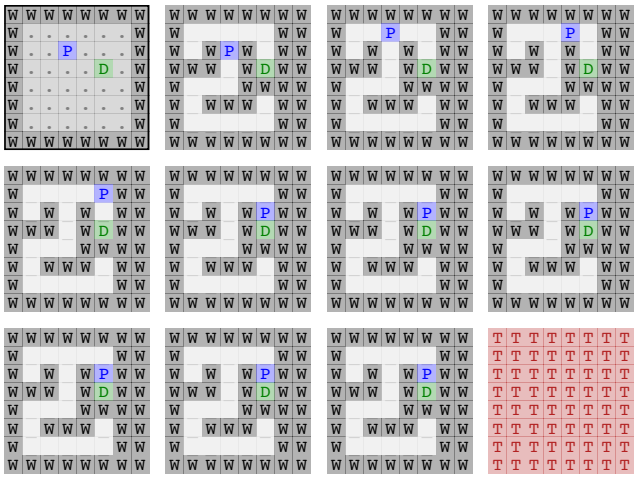

Figure 8: Generated $10T \times 6Y \times 6X$ **Maze** demonstrating the maze movement mechanic. The top left is the first timestep of the solution before grid population.

Grid Size	Success	Time	Extra P
$6T \times 4Y \times 4X$	0.60	$51m10s$	0.70
$7T \times 4Y \times 4X$	0.80	$41m12s$	0.60
Grid Size	**Eff. Length(*)**		**Trivial(*)**
$6T \times 4Y \times 4X$	3.5(6.0)		0.67(0.00)
$7T \times 4Y \times 4X$	2.9(4.0)		0.75(0.33)

Table 3: Results for **Sokoban**. (*) denotes results with levels containing extra Ps removed.

also show the ending condition that every B must be pushed onto a O (a block on top of a target is represented by G). No generated **Sokoban** levels include any of the wall Ws present in one of the example levels (Figure 5).

Limitations

Our experiments show that, while promising, this approach carries substantial limitations. First, our comparisons of **Sokoban**, **Maze**, and **Field** show that small increases in input size, output size, or game complexity can yield much slower solution generation times. We theorize that the input size increase causes STWFC to take longer because when propagating, there are more patterns available, causing fewer cells to "collapse" into a single pattern on each round and resulting in more rounds. A larger output size causes STWFC to take longer both because of the increased number of cells to populate, and because a larger output has more potential paths to the solution, again reducing the number of cells "collapsed" by propagation. Paradoxically, we suspect that more complex games take longer to generate because propagation affects more cells due to far-reaching changes.

Secondly, generated levels are often trivial, reaching a solution in two or fewer timesteps from the initial state. We hypothesize that this is due to the complex nature of generating level solutions. Longer solutions have more potential paths and are more likely to reach a state from which a solution is no longer possible, causing WFC to fail. When a solution is found early, on the other hand, the problem is reduced to only filling out the static portion of the level. One way to force longer solutions would be to not include pauses like the extra copies of the solution state in the example data that we used (discussed in **Experiments: Configuration: Padding**); this would require solutions to use exactly the desired number of timesteps to complete and could result in more failed generations.

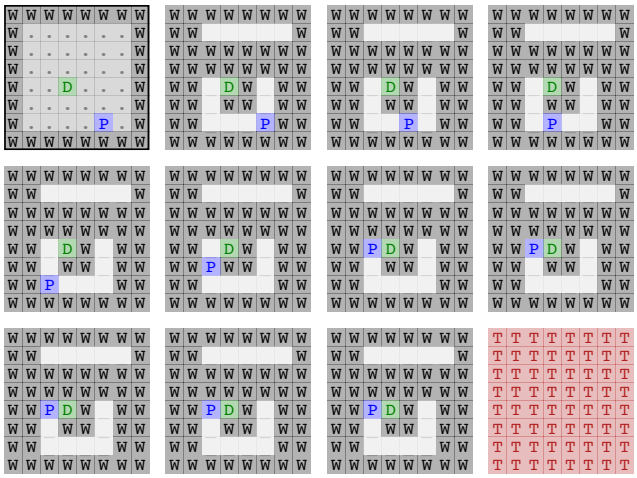

Figure 9: Generated $10T \times 6Y \times 6X$ **Maze** with an inaccessible region. The top left is the first timestep of the solution before grid population.

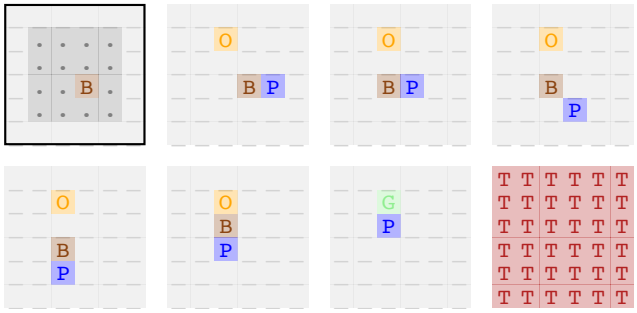

Figure 10: Generated $6T \times 4Y \times 4X$ **Sokoban** demonstrating the high-level block moving mechanic. The top left is the first timestep of the solution before grid population.

Additionally, we found that example level solutions fail to capture some important high-level constraints. Even after designing our example solutions to guarantee that all changes occur over a sufficiently small space-time window to be captured by our chosen pattern size, all three example games sometimes generate levels that violate constraints (most commonly by including an extra player object), making them invalid.

Discussion

Our results show that with very little example data, STWFC can generate small valid level solutions demonstrating

learned game mechanics and ending conditions of simple games. The advantage to this approach is that because the solution is generated along with the level, it is easy to confirm that a generated level not only looks like the examples but is actually completable. In addition, very little game-specific information needs to be included beyond example level solutions that demonstrate the game's mechanics.

Future work on STWFC should attempt to improve its efficiency and investigate ways of learning higher level space-time constraints from example solutions. One potential approach is to add annotations to the space-time data, taking inspiration from the practice of annotating 2D levels with path information. This approach could also be extended to work for 3D games with a 4th time dimension. Another possible extension would be to attempt to use patterns which are larger in the time dimension to capture game objects' acceleration. Finally, STWFC could be used to check the work of other level generators by finding solutions to 2D levels passed as input.

Conclusion

Existing approaches to improving the solvability of WFC-generated levels typically require adding additional game-specific information in the form of global constraints, substantially increasing the complexity and time required for setup. We have implemented a novel *space-time* WFC that uses both spatial and temporal dimensions of example level solutions as input. Experiments using this method show that STWFC is capable of generating level solutions that demonstrate learned game mechanics and ending conditions. However, levels are slow to generate, and even simple games still have high-level constraints that STWFC fails to capture.

Appendix

Remaining example levels used as input (Figure 11, Figure 12, Figure 13).

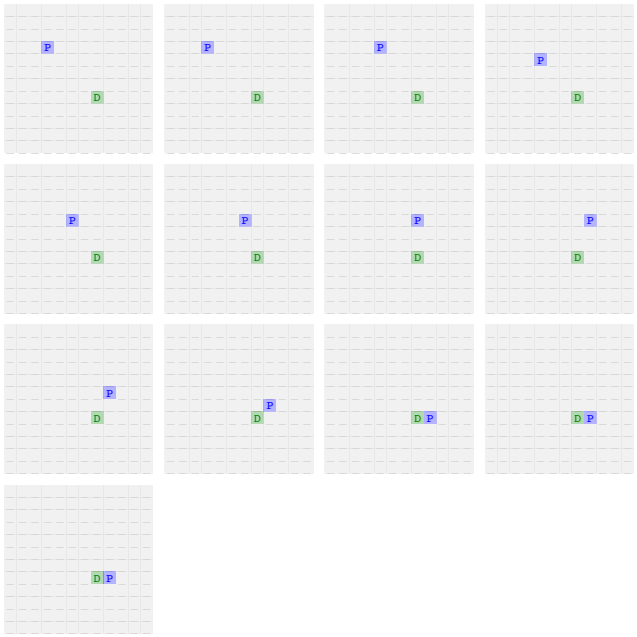

Figure 11: *Field 2* example level

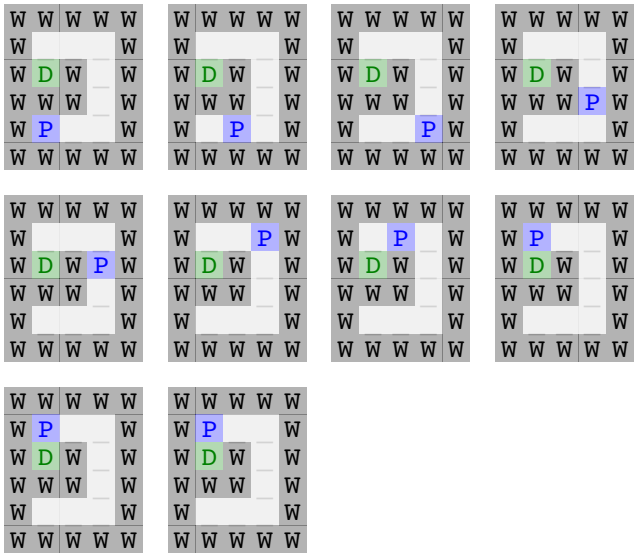

Figure 12: *Maze 1* example level

References

Alaka, S.; and Bidarra, R. 2023. Hierarchical Semantic Wave Function Collapse. In *Proceedings of the 18th International Conference on the Foundations of Digital Games*, FDG '23. New York, NY, USA: Association for Computing Machinery. ISBN 9781450398558.

Babin, M.; and Katchabaw, M. 2021. Leveraging Reinforcement Learning and WaveFunctionCollapse for Improved Procedural Level Generation. In *Proceedings of the 16th International Conference on the Foundations of Digital Games*, 1–8.

Beukman, M.; Ingram, B.; Liu, I.; and Rosman, B. 2023. Hierarchical WaveFunction Collapse. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 19, 23–33.

Cheng, D.; Han, H.; and Fei, G. 2020. Automatic generation of game levels based on controllable wave function collapse algorithm. In *Entertainment Computing–ICEC 2020: 19th IFIP TC 14 International Conference, ICEC 2020, Xi'an, China, November 10–13, 2020, Proceedings 19*, 37–50. Springer.

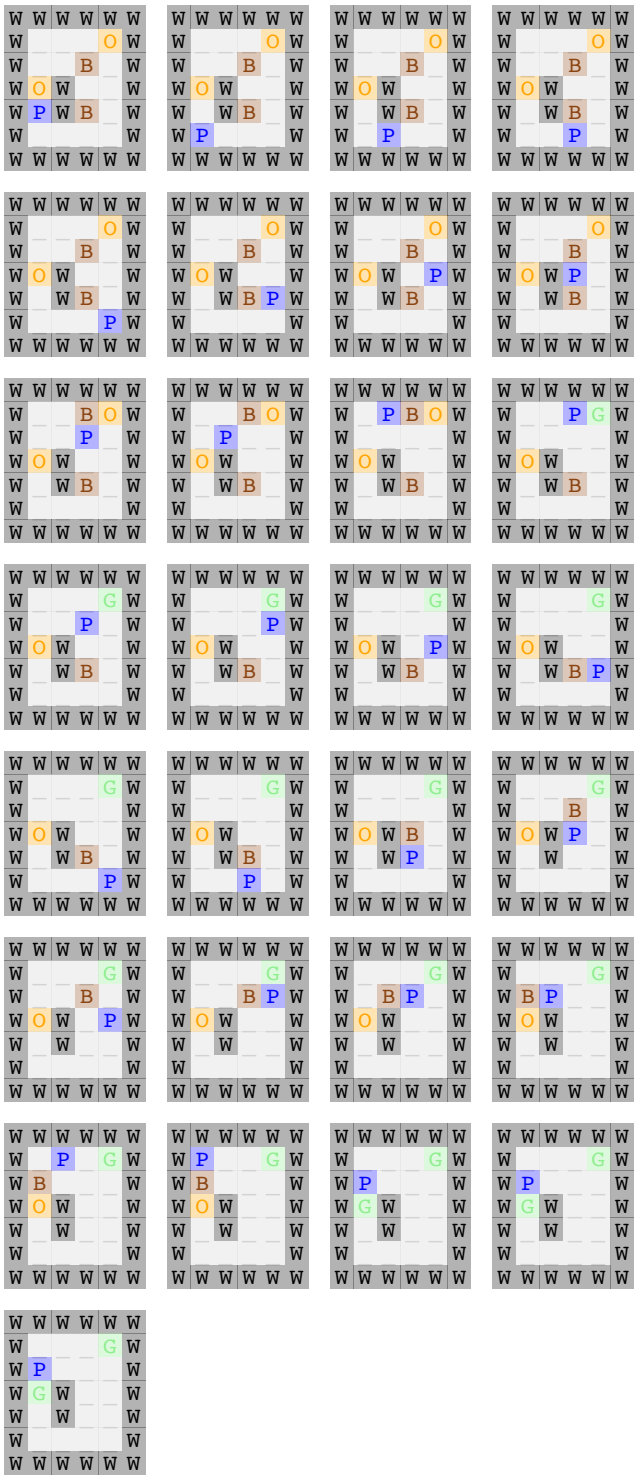

Figure 13: *Sokoban 2* example level

Cooper, S. 2022. Sturgeon: tile-based procedural level generation via learned and designed constraints. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 18, 26–36.

Cooper, S. 2023a. Sturgeon-GRAPH: Constrained Graph Generation from Examples. In *Proceedings of the 18th International Conference on the Foundations of Digital Games*, FDG '23. New York, NY, USA: Association for Computing Machinery. ISBN 9781450398558.

Cooper, S. 2023b. 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*, 1–9.

Cooper, S.; and Sarkar, A. 2020. Pathfinding Agents for Platformer Level Repair. In *AIIDE Workshops*.

Font, J. M.; Izquierdo, R.; Manrique, D.; and Togelius, J. 2016. Constrained Level Generation Through Grammar-Based Evolutionary Algorithms. In Squillero, G.; and Burelli, P., eds., *Applications of Evolutionary Computation*, 558–573. Cham: Springer International Publishing. ISBN 978-3-319-31204-0.

Gumin, M. 2016. WaveFunctionCollapse. https://github.com/mxgmn/WaveFunctionCollapse. Accessed: 2024-07-03.

Gumin, M. 2022. MarkovJunior, a probabilistic programming language based on pattern matching and constraint propagation. https://github.com/mxgmn/MarkovJunior. Accessed: 2024-07-03.

Horswill, I.; and Foged, L. 2021. Fast Procedural Level Population with Playability Constraints. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 8, 20–25.

Kartal, B.; Sohre, N.; and Guy, S. 2016. Data driven Sokoban puzzle generation with Monte Carlo tree search. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 12, 58–64.

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. 2022. WaveFunctionCollapse: Content Generation via Constraint Solving and Machine Learning. *IEEE Transactions on Games*, 14(3): 364–376.

Kim, H.; Lee, S.; Lee, H.; Hahn, T.; and Kang, S. 2019. Automatic Generation of Game Content using a Graph-based Wave Function Collapse Algorithm. In *2019 IEEE Conference on Games (CoG)*, 1–4.

Kim, H.; Seo, B.; and Kang, S. 2024. Puzzle-Level Generation with Simple-tiled and Graph-based Wave Function Collapse Algorithms. *IEEE Transactions on Games*, 1–11.

Le, V. 2019. wave-function-collapse. https://github.com/Coac/wave-function-collapse. Accessed: 2024-07-03.

Lee, V.; Partlan, N.; and Cooper, S. 2020. Precomputing Player Movement in Platformers for Level Generation with Reachability Constraints. In *AIIDE Workshops*.

Nie, Y.; Zheng, S.; Zhuang, Z.; and Togelius, J. 2024. Nested Wave Function Collapse Enables Large-Scale Content Generation. *IEEE Transactions on Games*, 1–11.

Rix, M. 2017a. 2 second loop. https://x.com/MattRix/status/872648369625325568. Accessed: 2024-07-03.

Rix, M. 2017b. 4 second loop. https://x.com/MattRix/status/872641331956518914. Accessed: 2024-07-03.

Rix, M. 2017c. another 4 second loop! https://x.com/MattRix/status/872645716660891648. Accessed: 2024-07-03.

Rix, M. 2017d. last one for tonight, a 3 second loop! https://x.com/MattRix/status/872674537799913472. Accessed: 2024-07-03.

Rix, M. 2017e. Using 3D Wave Function Collapse to create patterns that loop in X, Y, and Time. https://x.com/MattRix/status/872884946918150145. Accessed: 2024-07-03.

Rix, M. 2018. I don't think I ever posted this, but I also played around with having creation + destruction steps (which of course always end up with an equal amount of births vs deaths). https://x.com/MattRix/status/979020989181890560. Accessed: 2024-07-03.

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

Sarkar, A.; and Cooper, S. 2020. Sequential segment-based level generation and blending using variational autoencoders. In *Proceedings of the 15th International Conference on the Foundations of Digital Games*, 1–9.

Smith, A. M.; Andersen, E.; Mateas, M.; and Popović, Z. 2012. A case study of expressively constrainable level design automation tools for a puzzle game. In *Proceedings of the International Conference on the Foundations of Digital Games*, FDG '12, 156–163. New York, NY, USA: Association for Computing Machinery. ISBN 9781450313339.

Smith, G.; Treanor, M.; Whitehead, J.; and Mateas, M. 2009. Rhythm-based level generation for 2D platformers. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, FDG '09, 175–182. New York, NY, USA: Association for Computing Machinery. ISBN 9781605584379.

Smith, G.; Whitehead, J.; and Mateas, M. 2011. Tanagra: Reactive Planning and Constraint Solving for Mixed-Initiative Level Design. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3): 201–215.

Snodgrass, S.; and Ontanón, S. 2016. Controllable Procedural Content Generation via Constrained Multi-Dimensional Markov Chain Sampling. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, 780–786.

Snodgrass, S.; and Ontanón, S. 2017. Player movement models for platformer game level generation. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, 757–763.

Sorochan, K.; Chen, J.; Yu, Y.; and Guzdial, M. 2021. Generating Lode Runner Levels by Learning Player Paths with LSTMs. In *Proceedings of the 16th International Conference on the Foundations of Digital Games*, 1–7.

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.