### Title

WaveFunctionCollapse: Content Generation via Constraint Solving and Machine Learning




### Journal

IEEE Transactions on Games, PP(99)

### ISSN

2475-1502



### Authors

Karth, Isaac

Smith, Adam Marshall



### Publication Date

2021

### DOI

10.1109/tg.2021.3076368

Peer reviewed

# WaveFunctionCollapse: Content Generation via Constraint Solving and Machine Learning

Isaac Karth and Adam M. Smith, *Member, IEEE*

***Abstract*—We describe WaveFunctionCollapse (WFC), a new family of algorithms for content generation. WFC was recently invented by independent game developer Maxim Gumin and has since been adopted and adapted by other game developers. Trends in academic research on content generation have only recently suggested the use of ideas from constraint solving and machine learning, so it is surprising to see these manifested in in-the-wild algorithms developed outside of an academic context. We illuminate the common components in this family of algorithms by way of a rational reconstruction. Through experiments with the reconstruction we probe the impact of design choices made in various adaptations of WFC (e.g. the role of backtracking, search heuristics, or pattern classification and rendering strategies). This work highlights a mode of incremental content generation that has been overlooked by past surveys of content generation methods.**

***Index Terms*—Procedural Content Generation, constraint solving, WaveFunctionCollapse, rational reconstruction, search heuristics**

## I. Introduction

MULTIPLE academics have proposed ways to use constraint solving and machine learning for procedural content generation [1], [2]. However, procedural content generation, as it is practiced and studied, tends to focus on constructive and search-based techniques. In this article, we illuminate a surprising example of constraint satisfaction problem (CSP) and machine learning (ML) techniques: WaveFunctionCollapse (often known simply as WFC).

In the fall of 2016, a new approach to procedural content generation burst onto the scene. Invented by Maxim Gumin, WFC uses constraint solving and machine learning to translate minuscule artist-created training images into large, expressive generated content. It has been used on everything from small-scale game jam and PICO-8 fantasy console projects to larger commercial games released on the Nintendo Switch,[1] and has since been taught to undergraduates at several universities and used in technical games research.

This article is intended to be the definitive academic account of the emergence of WFC, continuing the examination started in 2017 with a workshop paper [3]. We present a compact overview of the family of algorithms that make up WFC, including the terminology and concepts that it uses or introduces, such as tiles, patterns, propagation, and the use of inputs and outputs in the algorithm's pipeline. This paper is intended to provide equal weight in its tutorial, technical, and survey contributions. For tutorial content, we present an approachable high-level pseudo-code walkthrough of the operation of WFC. For technical contribution, we include a rational reconstruction of WFC, which we use as a testbed for experiments on the contributions of the different elements in its pipeline. This additionally reveals how constraint solving and machine learning (including deep machine learning) can factor into the algorithm. Finally, we trace the adoption of WFC across academic, industrial, and artistic users.

### A. Research Context

WFC can be positioned as a development of a number of lines of previous research. While this does not lessen the surprise of its viral adoption, it does provide context.

*1) Texture Synthesis:* In computer graphics, *texture synthesis* is the problem of generating a large (and often seamlessly tiling) output image with texture resembling that of a smaller input image [4]. In many texture synthesis approaches (e.g. the work of Liang et al. [5]), the input and output images are characterized in terms of the local patterns they contain, where these patterns are typically sub-images of just a few pixels in width (e.g. 5-by-5 pixel windows). Many texture synthesis algorithms explicitly intend to produce outputs such that every local pattern in the output resembles a local pattern in the input. In the visual setting of graphics, this resemblance need not be exact pixel-for-pixel matching and is often judged based on a distance metric (e.g. Euclidean distance of pixel color vectors) that judges some colors to be closer than others. By contrast, exact matching is the only sense of resemblance present in WFC.

In Liang's method [5], the output image is grown incrementally. Part-way through the generation process, a large region of the output has already been generated, but more remains. A location on the border of this region is selected, and the surrounding already-chosen pixels (the context) are used to query an index of patterns generated from the source image. A pattern with similar local pixels is retrieved and used to paint in a few more pixels of the output image, growing the region of completed pixels. WFC also grows its output image incrementally, expanding the known regions of the output by completing them with details from local patterns of the input image. However, it also considers the not-yet-known space because of its consideration for exact pattern matching.

While WFC is loosely inspired by quantum mechanics,[2] Gumin was also inspired by Paul Merrell's discrete synthesis

I. Karth and A.M. Smith are with the Computational Media Department, University of California Santa Cruz, Santa Cruz, CA 95064 USA email: ikarth@ucsc.edu and amsmith@ucsc.edu

[1]Respectively: https://arcadia-clojure.itch.io/proc-skater-2016, https://trasevol.dog/2017/09/01/di19/, and *Bad North* (2018)

[2]Very loosely, and mostly confined to how it models the superposition of possible image states. As Gumin explains, "The coefficients in these superpositions are real numbers, not complex numbers, so it doesn't do the actual quantum mechanics, but it was inspired by QM." [6]

approach. While Merrell was inspired by texture synthesis, he focused on the problem of generating 3D geometric models [7]. In this setting, we want to generate a new (typically large) 3D model which is made up of components and arrangements taken from a (typically small) 3D model provided by a human artist. Per texture synthesis traditions, artifacts are characterized in terms of their local patterns on a regular grid. Instead of blendable pixel colors, however, discrete model synthesis aims to exactly reuse rigid geometric chunks.

In personal correspondence with us, Gumin described how he was influenced by convolutional neural network style transfer, but found it lacking for videogame level generation. He experimented with several approaches to model and texture generation, looking for a texture synthesis algorithm with strong local similarity, where each $N \times N$ pattern in the output could be traced to a specific input pattern. Gumin's intent was to capture the rules for how the source image was made.

Gumin's SynTex project [8] implemented several texture synthesis methods, yielding attractive results for game texture images but nonsensical outputs for non-texture images where pixel-grid analysis destroyed the visual semantics of structured objects (e.g. swords). In his ConvChain project [9], he experimented with an approach based on Markov Chain Monte Carlo (MCMC), a statistical sampling approach that directly measures how likely an output image is under the distribution of local patterns implied by the input image. Statistical modeling is also present, if much less explicitly, in the notion of entropy used in Gumin's later WFC algorithm.

*2) Constraint Solving:* In the field of artificial intelligence (AI), largely disconnected from computer graphics until recently,[3] constraint solving uses ideas from knowledge representation and search to model continuous and combinatorial search and optimization problems and solve them with domain-independent algorithms [11, Chap. 6].

Constraint satisfaction problems (CSPs) are typically defined in terms of decision variables and values. In the context of WFC-style image generation, there is a variable associated with each location in the output image. In a solution to the problem (called an assignment), each variable takes on a value. Depending on the context, values may come from continuous or discrete domains. For the task addressed by WFC, the values are associated with the discrete set of unique local patterns in the input image. The choice to assign a specific variable a specific value will often constrain the available choices that can be made for other variables. Constraints relate the legal combination of values that a set of variables might take on in a valid assignment. For the image generation task, we want to model the idea that the patterns chosen at each location in the output are compatible in terms of exact matches for the pixels in which their associated local windows overlap.

The goal of an algorithm for solving CSPs (a solver) is to find a total assignment (an assignment for every variable) such that no constraints are violated. Although there are many different approaches to constraint solving, most operate by searching in the space of partial assignments. That is, they search the space of incomplete solutions, not generating a single candidate solution until that solution is known to be free of conflicts (constraint violations). The solver repeatedly *selects* an unassigned variable and then *decides* on a value to assign from the variable's domain. If the solver encounters a partial assignment for which no subsequent variables can be assigned without violating constraints, the solver typically backtracks on a recent decision—backing out of a dead-end by moving to another point in the space of partial assignments.

To the skeleton of backtracking search sketched above, advanced constraint solving methods add improvements that attempt to speed up identification of a legal total assignment. Some *heuristics* (either domain-specific or domain-independent) aid the selection of a promising variable to select next while others aid the decision of a promising value to assign for that variable. The addition of heuristics typically alter the order in which the solver explores the space without impacting completeness guarantees (i.e. that the solver will return a solution in bounded time if at least one exists).

Complementary to heuristics, *constraint propagation* methods do additional bookkeeping in order to prune away values from domains that would lead to dead-ends later. Constraint propagation ideally allows a solver to skip past fruitless search without impacting the order in which the space is explored. AC-3 is a well known constraint propagation algorithm [11, Chap. 6]. Although AC-3 and other propagators can end up making assignments to variables as part of their operation, they are not complete solvers by themselves. Propagators are typically run after each choice by a solver in order to simplify the remaining search problem. For a game-focused audience, we refer the reader to the *Game AI Pro 2* book chapter "Rolling Your Own Finite-Domain Constraint Solver" [12] for more details.

*3) Constraint Solving in PCG:* Although there are a few examples of note, until recently constraint solving was mostly overlooked for the purposes of content generation. Taxonomies of PCG such as in the notable search-based PCG survey [13] do not account for approaches to content generation that are neither directly constructive nor perform their search at the level of completed candidate designs. The concept of working with partial designs is part of what makes the animations derived from WFC executions so visually stunning—we are not used to seeing our generators work this way.

Constraint-based PCG methods are often associated with making strong guarantees about outputs as well as having the cost of those guarantees paid in unpredictability of total running time. Most backtracking solvers yield good performance on their associated search tasks for real world problems, but this outcome is hard to characterize in terms of theory (where exponential worst case analyses are uninformative). Horswill and Foged [14] describe a "fast" method for populating a level design with content under strong playability guarantees. Their algorithm is based on backtracking search with (AC-3) constraint propagation. Although it makes only modest demand on processor and memory resources, it is expected to be used by programmers who are at least moderately literate in search algorithm design.

[3]Recent innovations in style transfer were sparked by a breakthrough in using deep convolutional neural network classifiers to mimic artistic styles [10] This has led to a flood of related research, along with the exploration of other applications of neural networks to graphics.

In G. Smith's Tanagra system [15], a mixed-initiative platformer level design tool, the Choco [16] solver is invoked to solve a specific geometric layout subproblem in the overall level design process. In this system, the user is in a designer role rather than a programmer role. When the solver determines that the given CSP is impossible to solve (we say the constraints are unsatisfiable), it signals to the larger tool that other decisions about the working level design, such as what activity the player performs on each platform, need to be relaxed (backtracked). Although Tanagra illustrates that CSPs need not only be created by programmers (they can be assembled programmatically from the data input into a graphical user interface), backtracking still plays a major role. By surprising contrast, Gumin's original formulation of WaveFunctionCollapse does not make use of backtrack.

*4) ASP in PCG:* Answer set programming (ASP) is a form of logic programming targeted at modeling combinatorial search and optimization problems [17]. In ASP, low-level constraints are automatically derived from the high-level rules in a problem formulation program, and the implied CSP is solved using algorithms rooted in the SAT/SMT literature [18].

A. M. Smith proposed the use of ASP in PCG [1] within the paradigm of modeling design spaces. Rather than directly aiming to code and algorithm for generating content, this approach suggests we should declaratively model the space of content we want to see and let a domain-independent solver take care of the procedural aspects. Although programmers using ASP need not have or use any knowledge of search algorithm design, they are expected to be familiar with the declarative programming paradigm and Prolog-like syntax. This background is not common amongst technical artists who were recently excited to find WaveFunctionCollapse.

Modern answer set solvers (such as Clingo [19]) allow for specification of custom heuristics, externally checked constraints interleaved with the search process, and hooks for scripting languages in the service of integrating solvers with outside environments. These custom extension points in Clingo have been used to replicate the incremental growth behavior of WFC to produce an animated[4] log of the solver's decisions.

## II. Illuminating Gumin's Reference Implementation of WFC

In this section, we examine the general idea of Gumin's original formulation of the WFC algorithm [20]. The details of the generation process–and observed variations–will be discussed further below, while this overview will provide a high-level introduction.

Although Gumin's project (including utilities for generating the example animations that attracted so many others to WFC) is not large—it involves less than a thousand lines of C# code—the broad ideas of the algorithm are difficult to interpret by directly reading the code. In personal correspondence and observing several users of WFC online, we learned that many of them treated the code as a black box, either using it directly without attempting to alter it or re-implementing their own versions based on the animations of WFC.[5] In response, we offer a pseudocode summary below.

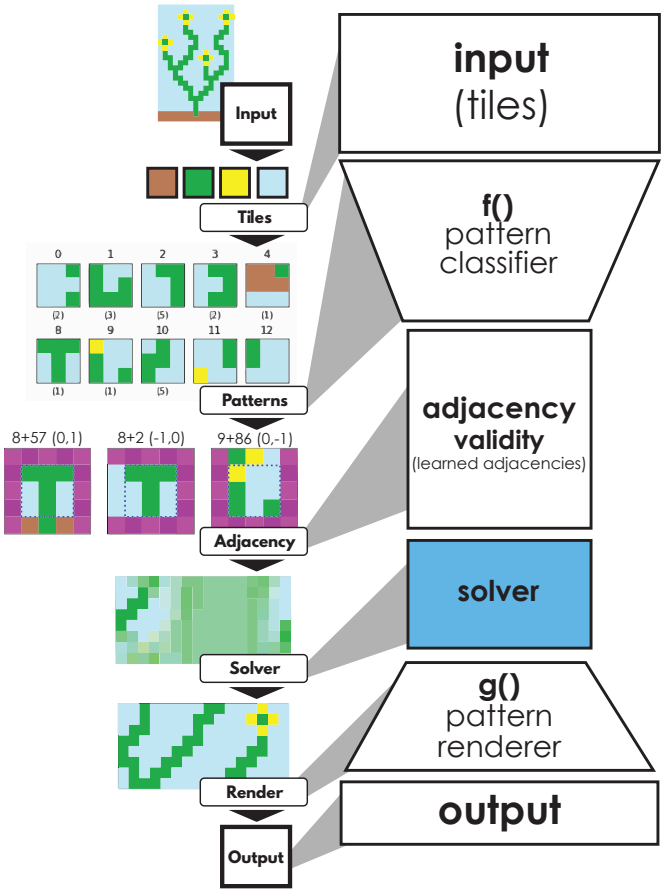


Fig. 1. Diagram of the WFC generative pipeline, using the overlapping model to automatically determine allowed adjacencies between tiles. Left column shows selections of data at each step in the pipeline, right column shows how the functions in the pipeline are related. Starting with a source image, it is (1) subdivided into tiles; (2) tiles in source image are classified into patterns via a convolution of their local neighborhood; (3) a matrix of pattern adjacency constraints is constructed based on how patterns overlap with neighboring tiles, the image here shows how the two neighboring patterns overlap on a specific dimension; (4) the constraint solver generates a new configuration of patterns; (5) the pattern renderer converts the patterns into tiles (or pixels or whatever other representation is desired); (6) finally, resulting image is output.

One way that WaveFunctionCollapse stands out from many other PCG algorithms is that both the input and the output are images. While there are many generative operations that can act on images (e.g. as a filter) there are comparatively few that take an image as constraint specification. At a high level, the processes of WFC analyze the input image, solve the constraints found by the analysis over a grid of user-specified size, and render the result as an image again (Fig. 1). Many machine learning image generation algorithms require millions of images in training data. In contrast, the training data for WFC is tiny: a single image, often less than 32x32 pixels.

The WFC solver typically operates on $N \times N$ *patterns* of tiles rather than on individual tiles, referred to as the

[4]"Visualizing WaveFunctionCollapse reimplemented in ASP: 46x46 Flowers N=3, using clingo's default heuristic (VSIDS). Just 3 conflicts." https://twitter.com/rndmcnlly/status/867605489789489152

[5]"It was hard to crack your computer science lingo about collapsing wave functions, so I basically just looked at your gifs" https://twitter.com/OskSta/status/784850280093478912

*overlapping pattern model*. Gumin characterizes the relationship between a simple tile model and the overlapping pattern model as being the same as the relationship between higher order Markov chains and order-one Markov chains [6]. The implications of using patterns will be examined in more detail below, but for now the important property to note is that they can be interpreted as *convolutions* that compress the surrounding context into a value associated with a single point. WFC is primarily concerned with grids of these pattern identifiers. The final result of the analysis is a set of patterns and an adjacency matrix that describes which patterns can be validly placed next to one another.

Listing 1. Basic WFC Constraint Solver

```
function Solve(adjacencies, wave_matrix):
  loop while no contradiction:
    wave_matrix := Propagate(wave_matrix,
      ↪ adjacencies)
    if all domains collapsed:
      return wave_matrix
    wave_matrix := Observe(wave_matrix)
  throw error: generation attempt failed
```

The generation process seeks to fill in the details for a new grid of patterns (the shape of the new grid does not necessarily match the shape of the grid seen in the input). During the core generation loop, the algorithm solves the constraint problem implied by the pattern adjacency matrix. It repeats a two step process (contextualized in Fig. 2): `Observe(grid)`, which finds the most constrained variable and picks a value to assign via weighted random sampling, and `Propagate(grid, adjacencies)`, which updates the domains of variables at each grid location.

Listing 2. WFC Observation Step

```
function Observe(wave_matrix):
  # find the minimum entropy cell in grid
  cell = FindMinimumEntropy(wave_matrix)
  If any cell has zero possibilities:
    # this is contradictory state, where a node has
      ↪  an empty domain
    return paradox failure exception
  If all cells have exactly one possibility:
    return wave_matrix
  In node with the least entropy (ties broken
      ↪ randomly):
    Assign a value via weighted random sample of
      ↪ the cell's current domain.
    Add node to the propagation stack.
```

Gumin's implementation of WFC uses a metaphor of *entropy* to identify the most constrained cell (least risk choice) at each stage. This notion is similar to, but not identical to, the notion of entropy of probability distributions in statistics: both are maximized for a uniform distribution (when anything is possible) and minimized for the distribution with only one option (when there is no uncertainty about the result). Gumin defines the entropy of a cell as the weighted sum of patterns that may still be validly placed at that cell (with weights based on how often that pattern was seen in the input image). The same weights are used when deciding which pattern to place at the selected cell. This explicitly random search is not a usual default approach to constraint solving, though many solvers support it as a configurable option. While a solution can be found without using the weighting, the weighted random search is important for Gumin's secondary goal for local similarity: the distributions of patterns seen in the output should be similar in the input and output [6].

If the entropy of a cell reaches zero, with no valid members in its domain, the solution is in a contradictory state. Rather than using local backtracking, Gumin's implementation simply globally restarts when it reaches a contradiction.

Once a cell is solved (the domain of the associated variable is reduced to a single value), WFC propagates the implications of the change to the neighboring cells. Like AC-3, WFCs propagation procedure implements *arc consistency*—it ensures that a value only appears in a domain of a variable if there exists a valid value in the domain of related variables such that constraints over those variables could be satisfied. When originally released in 2016, Gumin's code implemented an AC-3 style propagation algorithm that was later improved with an asymptotically more efficient AC-4 style algorithm in 2018.

Listing 3. WFC Propagation Step.

```
function Propagate(wave_matrix, adjacencies):
  Initialize the propagation stack to contain the
      ↪ last cell observed.
  while there are cells on the propagation stack:
    for neighbor of this cell:
      for neighbor_pattern in neighbor_domain(
          ↪ neighbor):
        if not( adjacency(original_pattern,
            ↪ neighbor_pattern) ):
          Decrement count of neighbor_pattern.
          if count is zero:
            Remove neighbor_pattern from
              ↪ neighbor_cell.
            Add neighbor_cell to the propagation
              ↪ stack.
  return wave_matrix
```

## III. Rational Reconstruction

Now that we have introduced a high-level summary of the reference implementation, we can break it down further, into modular substages. WFC is not a monolithic generator, but rather a multi-stage pipeline of generators and data transformations (Fig. 2). Many variations on this pipeline exist in the wild, but all of the members of this family share the core adjacency-learning and constraint solving stages. Breaking the general case down to component steps, the stages in the complete list are: making data input interpretable as tiles, tile classification and grouping, pattern classification and adjacency learning, constraint solving, and rendering the result of the constraint solver into tile data. The description that follows is intended to be detailed enough to guide a reader in producing their own implementation of WFC.[6].

### A. Adjacencies

Conventionally, WFC operates on a rectilinear grid. This gives us an easy way to define the cells and adjacency edges: the cells are each intersection on the grid and the edges are the lines between them. However, WFC can work on any graph for which a well-defined adjacency function can be specified.

[6]Our Python reconstruction can be found at https://github.com/ikarth/wfc_2019f

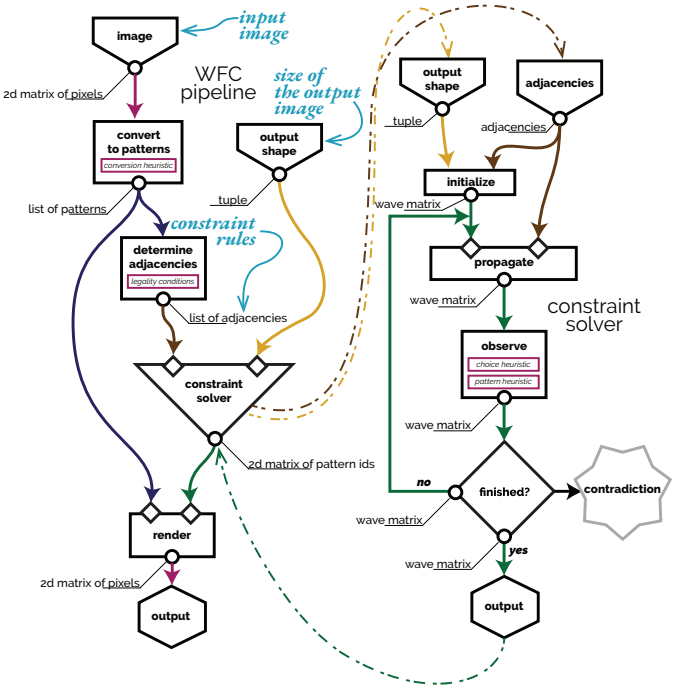


Fig. 2. Diagram of data and control flow in the WFC generative pipeline. Given an image, WFC finds patterns within it and then determines the valid adjacencies between those patterns. These become the constraint rules for the solver—one of the two main data type it uses. The other major data in the solver is the wave matrix, a Boolean representation of all of the pixels in the image indexed against all of the patterns. The solver alternates between propagating the implications of the constraint rules (propagate) and constraining a domain via assignment (observe).

For example, Stålberg's experiments with triangular meshes[7] or Florian Drux's GraphWaveFunctionCollapse.[8] These can be non-spatial: to create a looping animation Matt Rix added a time edge.[9] In fact, Gumin notes that $d$-dimensional WFC can capture the behavior of any $(d-1)$-dimensional cellular automata [6]. Similarly, Martin O'Leary's WFC-based poetry generator[10] uses non-local edges for rhyming patterns and scansion. We call the basic unit that patterns are constructed out of *tiles*. Tiles, in our terminology, are the atomic units of the input data: the palette of pixels in most of Gumin's WFC examples, the set of game map tiles in *Caves of Qud*, the collection of modules of 3D geometry in *Bad North*, the lexicon of words in O'Leary's *Oisín*, and so forth. Generalized, tiles are discrete sets of items that can be uniquely recognized at each location in the input and for which new outputs can be assembled from a grid (or graph) of tile identifiers.

Because the constraint solver is the heart of WaveFunctionCollapse, the adjacency constraints are critical data and the adjacency learning stage is a vital (but distinct) part of the overall generator. In the most extreme simplification these can be the literal input tiles, with hand-written adjacency data. However, rather than operating on literal tiles, WFC typically operates on higher-order data about the tiles plus their local context: Gumin's name for this data is *patterns*. Other strategies for learning adjacency include the approach used in Bad North, which takes the 3D geometry modules (exported from a 3D modeling program) and classifies them by whether the profiles match. Profiles are the (polyline) cross-sections of 3D meshes sliced at grid cell boundaries.

Gumin's OverlappingModel describes patterns as an $N \times N$ region of tiles (where $N$ is typically 2 or 3)[11] that act as the context for a given tile. Therefore, a single tile can be in multiple patterns. Valid pattern-region adjacencies are when the intersection of two pattern-regions are identical when overlapped with a given offset in space (e.g. pattern 8 overlaps with pattern 57 in direction (0,1) in Fig. 1). Gumin's implementation represents this as a Boolean matrix with the shape *pattern* $\times$ *pattern* $\times$ *offset direction* which is optimized for quick lookups in the solver, though when pre-processing the patterns the list of adjacency tuples is often easier to work with.

The analysis can be considered a form of machine learning [21] Whether one pattern can be placed next to another with a given offset can be decided by a classifier that accepts two patterns and outputs a Boolean judgment. This classifier is fit to agree with the adjacencies demonstrated in the source image. One simple representation of this classifier is simply a list of allowed pattern pairings.

### B. Pre-constraints

Sometimes we want to further constrain the output. For example, Gumin's reference implementation has support for pre-constraining the bottom image row, which is useful for things like side views, such as the Flower sample. This can be done by simply setting the patterns in the domain of the nodes on the bottom row to only include patterns with dirt tiles and removing dirt tiles from the domains of all of the nodes in the sky. Generalizing this, we can supply a partially-painted image with some of the final tile values already in place while leaving gaps to be filled in by the solver. We expect the user to specify pre-constraints at the level of tiles so that they do not need to think about the notion of patterns used by the generator.

### C. Constraint Solving

The core of WFC is the constraint solver. This can be implemented with an existing constraint solver, such as Clingo [3]. However, Gumin's reference implementation is a custom-written probabilistic constraint solver in C#, which is tailored to the requirements of WFC constraint solving. The following description breaks down how each part of the solver in the reference implementation works, as well as some remarks on variations.

[7]"Content-agnostic algorithm for placing tiles. Heavily based on the work of @ExUtumno. Basically a Sudoku-solver on steroids." https://twitter.com/OskSta/status/784847588893814785

[8]https://github.com/lamelizard/GraphWaveFunctionCollapse

[9]"last one for tonight, a 3 second loop!" https://twitter.com/MattRix/status/872674537799913472

[10]https://github.com/mewo2/oisin

[11]Large values of $N$ rapidly increase the complexity of the generated patterns.

*1) Initialization with Clear():* The state of the solver is primarily captured in one data structure: the *wave matrix*, representing which patterns are still valid for each cell by a Boolean value. The *adjacency matrix*, representing the patterns that are valid for each edge per pattern, is unchanging in the solver. Optionally, an auxiliary table of counters may avoid having to recalculate the number of remaining patterns and their sampling weights. At the start of solving, every entry in the wave matrix cleared to a value of true.

One metaphor is to consider the wave matrix as a chess board, with puzzle pieces stacked in each space, representing the patterns. At the start each space on the board has a stack of all the patterns. *Propagation* is the process of removing pieces that no longer fit with any of their potential neighbors. *Observation* is the process of removing all but one of the puzzle pieces. If there is an empty square, a contradiction has occurred and the current state is invalid. The goal is to get the chessboard into a state where every square has exactly one puzzle piece.

*2) Removal of alternatives with Ban():* An operation that is shared by both observation and propagation is to remove patterns from the domains of the cells. The reference implementation has a separate function for this. It takes a grid location and a pattern and does the bookkeeping to remove the pattern from the constraint solving domains. The wave matrix entry for the location and pattern are set to false, the pattern is removed from the edge compatibility, and the entropy values are decremented. The neighboring nodes are added to the stack for updating during the propagation phase.

*3) Choice of Design with Observe():* The purpose of Observe() is to select a decision variable and decide its value. The reference implementation uses a heuristic that chooses the most constrained cell. The cell with the tightest or smallest domain after propagation has the lowest entropy, with ties broken at random. There are two termination cases: first, if any of the cells have zero remaining patterns, the search has dead-ended with a contradiction. Second, if all of the cells have an entropy of one (i.e. there is exactly one pattern for each node, which is the minimum non-contradictory state) we have found our solution. The heuristic of selecting the most constrained variable or equivalently the variable with minimum remaining values (MRV) is well known in the constraint solving literature [11, Chap. 6].

The strategy of selecting the location with the lowest non-zero entropy (or minimum remaining values) might seem arbitrary at first. If we want to optimize our chances of uncovering a total assignment without restarting or backtracking, we should make choices that maximize the number of total assignments consistent with our choices so far. This avoids ruling out (potentially extremely rare) legal total assignments under the assumption that they are distributed amongst other total assignments. If the number of remaining total assignments is approximated as the product of the size of the domains of the unassigned variables (in other words, assuming the remaining choices are independent), then assigning the location with the smallest domain (lowest entropy / minimum remaining values) maximizes the value of the product after the assignment. To make another loose physics connection (this time to statistical mechanics), Gumin's (least) entropy heuristic could be interpreted as an intent to maximize the entropy of a uniform distribution over possible completed designs.

Once a cell is chosen, one pattern needs to be selected from the domain. This is done through weighted random sampling, with the weight derived from the frequency that the pattern appears in the input training data image. This implements Gumin's secondary goal for local similarity: that patterns appear with a similar distribution in the output as are found in the input [6]. All non-selected patterns are then removed from the cell's domain. The reference implementation does this with the Ban() function which also adds the neighbors onto the propagation stack and takes care of the entropy bookkeeping.

The first observation selects between all of the possible patterns, but the subsequent observations tend to look at vastly fewer options. The exact number is heavily dependent on the input image, but it is typical that the algorithm only chooses between two or three remaining valid patterns in a domain.

*4) Resolving implications with Propagate():* Once an observation has been performed, the neighboring cells need to be updated. Updating the domain of one cell implies the need to update the adjacent cells, propagating the implications of the previous observations via the Ban() function.

Even within Gumin's reference implementation of WFC, the propagation approach has changed over time. Early pre-release implementations of the propagation function used belief propagation, but was later changed to "constraint propagation with a saved stationary distribution" [6] for performance reasons. A further post-release optimization changed the reference implementation of WFC to execute propagation with a stack-based flood fill, incorporating performance improvements introduced by Fehr and Courant [20], [22].

Propagation can result in the domain of a cell being reduced to one pattern, completely resolving that cell. The distribution of which cells are resolved by propagation and which through observation varies based on the input image, and visualizing this can be useful for debugging.

The observation-and-propagation loop demonstrates that WFC is a family of algorithms that use constraint solving as the core generation algorithm. Indeed, Gumin occasionally describes the algorithm as a constraint solver.[12] It uses the minimum remaining values (MRV) heuristic to select the variable to decide next. For decisions, it uses the heuristic of choosing patterns according to their distribution in the original image. An alternative to this heuristic would be to use the well known least constraining value (LCV) selection heuristic [11]. (LCV can also be motivated by the maximum entropy principle.) However, it is difficult to predict the implications of this heuristic choice for the purposes of content generation. The topic of sampling from combinatorial spaces with statistical uniformity guarantees is surprisingly subtle [23].

*5) Contradictions:* Unlike many constraint solvers, the reference implementation of WFC does not employ backtracking. When it encounters a contradiction, it restarts from the beginning, giving up if it fails too many times (by default, the

[12]"I can help with the grasp part. WFC is basically a constraint solver. You start with everything unknown and when possible..." https://twitter.com/ExUtumno/status/793601984800624640

limit is ten attempts). The nature of the generation task makes this less of an issue than it would be for a general constraint solving problem: the patterns are likely to fit together (at least in the same way they did in the input image), ideally in many different configurations, so for most input images in the sample set at the sizes the solver is trying to generate, contradictions are very rare.

It is possible to construct an input image that forces contradictions: the sample set has the example of a wrapping chessboard with odd length and width, which is, of course, impossible to solve. It is also possible to construct inputs that are trivially perfect.[13]

WFC encounters contradictions more frequently with more restricted tilesets and larger output spaces. They also, as discussed in our experiments below, can become more frequent when additional global constraints are introduced. At this point, the addition of backtracking becomes more important, as does the ability to distinguish between an impossible problem and a merely complicated one.

### D. Rendering

Once the constraint solver has discovered a solution, it is represented as a grid of pattern identifiers. Therefore, one final step needs to be performed to translate the patterns back into tiles. In the reference implementation this is straightforward: because the patterns overlap, each cell resolves to exactly one tile. In most WFC variations, tiles are simply reproduced in the form they appear in the source image.

However, variations are possible. One variation in the reference implementation is used to render partial solutions: because a partially-solved constraint problem has more than one pattern per cell, the visualization for the algorithm in progress averages the colors of the tiles derived from the patterns in the domain for each cell.

An implementation that uses a more interesting representation for the tiles might include additional processing in the rendering step. For example, if the tiles are part of a 3D landscape generator, the number of trees added on this particular tile can be additionally informed by how many other forest tiles are nearby, quite apart from the encapsulated WFC step in the generation pipeline. A generator that has a coastline might use only water and land tiles in the WFC patterns, and only add the coastal transition sub-tiles as part of a post-processing step. This possibility is examined more deeply in one of the experiments in the next section.

### E. Visualizing WFC

One of the factors that lead to WFC going viral[14] are the visualizations of the solving process. The way these animations visualize modifications to partially-completed design immediately sets this generative method apart from ones based on a generate-and-test paradigm where only complete designs are considered during execution.

In addition to the visualizations of the intermediate states of the solving (showing the averaged possibility space of each cell) other useful visualizations can help us understand what is happening or has happened during the generation process. We here introduce a visualization that we call a *crystal growth diagram* (see on the bottom row of Fig. 3), a map showing the order in which the solver resolved the final contents of each cell in a single image rather than an animation. Such diagrams record the incremental growth of a design without the use of animation in the same way as the patterns of colors apparent in the cross-section of a geode.

Other visualizations we have found useful for debugging are tracking whether a cell was resolved with propagation or observation, the number of patterns an observation selects from, and the number of remaining possible patterns for a cell. These visualizations have revealed that most cells are resolved via propagation, and that the cells with the least remaining patterns (those likely to be selected for observation next) are usually adjacent to cells that have been recently resolved.

## IV. Experiments

There are many implementations of WFC (the reference repository links to over 20 publicly available implementations in more than a dozen languages). While most function similarly, there are many variations already extant: adding a time dimension,[15] edges for rhymes and metre, [24][16] using backtracking, or adding global constraints (e.g. testing if a level is traversable from entrance to exit). This section experimentally probes the importance of design decisions in Gumin's original WFC, aiming to separate the essence of the idea from its first implementation. In particular we ask:

- Is the *entropy* location heuristic necessary?
- Is the *weighted* pattern heuristic necessary?
- Is it safe to run WFC *without backtracking* when plausible global constraints are added?
- Does WFC need to operate on *human-curated* tiles?

### A. Location (Selection) Heuristics Experiment

The heuristic Gumin used for selecting the location of observations in WFC is partially inspired by observing humans drawing: "Why the minimal entropy heuristic? I noticed that when humans draw something they often follow the minimal entropy heuristic themselves. That's why the algorithm is so enjoyable to watch."[17] However, we know that other heuristics are possible: several of the other implementations of WFC use different ones. Therefore, in our first experiment, we aim to understand the importance of Gumin's entropy heuristic. We do this by comparing multiple heuristics in our reconstructed Python implementation[18] which is equivalent to Gumin's in

[13] Gumin gives an example of an "easy" tileset: "the unrestrained knot tileset (with all 5 tiles being allowed) is not interesting for WFC, because you can't run into a situation where you can't place a tile" and without "special heuristics" the length of correlations in the generated image quickly fall [6].

[14] "Procedural generation from a single example by wave function collapse" https://twitter.com/ExUtumno/status/781833475884277760

[15] @MattRix, Jun 8, 2017: "yep exactly! :) it has 3 dimensions, but the trick is that it has 14 edges so it can check forward & backward "diagonally" in time" https://twitter.com/MattRix/status/872785320445706241

[16] https://github.com/mewo2/oisin

[17] https://github.com/mxgmn/WaveFunctionCollapse/blob/master/README.md

[18] https://github.com/ikarth/wfc_2019f

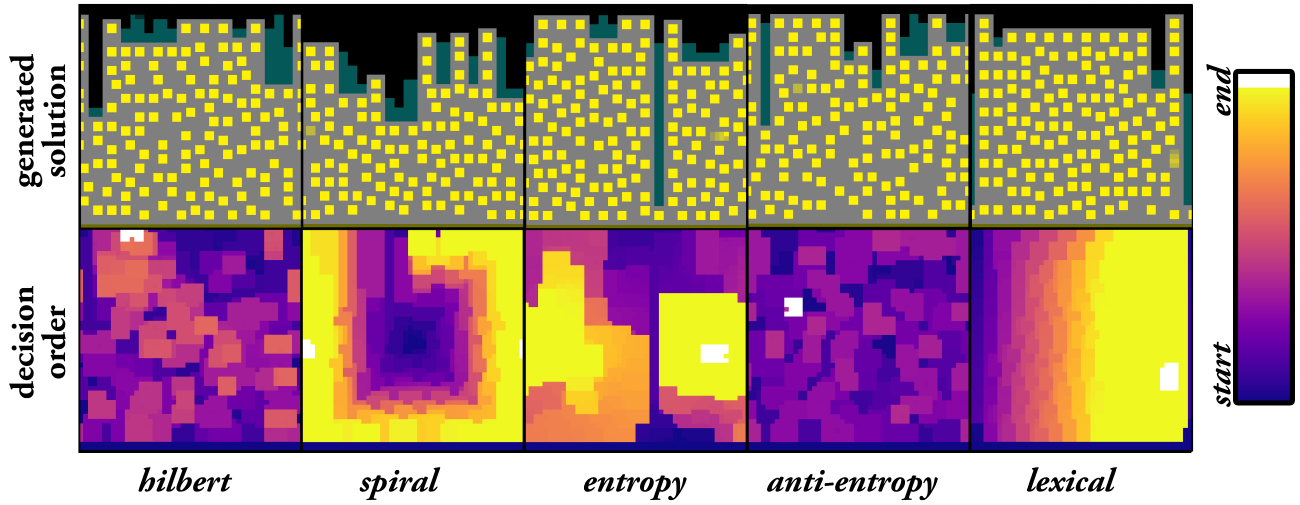


Fig. 3. Visualizing the order in which design decisions are made when using different heuristics. Final results are shown on the top row while crystal growth diagrams on the bottom show the order in which the result was assembled. Note that with strongly anisotropic pattern adjacencies (e.g. a side-view of a skyline) some location heuristics (such as *lexical*) tend to produce more clustered results (in this case, placing nearly all of the roofs of the buildings near the top of the generated image) which may or may not be aesthetically desirable.

terms of the state of the wave matrix after each observe–propagate cycle.

For all of the experiments, we used the reference implementation's selection of example source images. These define 53 different settings for testing the overlap WFC model. The heuristics tested were as follows:

**entropy** Original entropy calculation: select the node with the most constrained domain.

**anti** Opposite of entropy, select the node with the least constrained domain, breaking ties randomly (Included for a comparison as the most degenerate strategy).

**lexical** Select the first unsolved ($domain > 1$) node in top-to-bottom, left-to-right lexical order.

**random** Select a random unsolved node.

**spiral** Select the first unsolved node found by traversing a spiral path outward from the center.

**hilbert** Select the first unsolved node along a space-filling curve.

Rather than measuring wall-clock time (which is likely to vary between implementations), we measure the number of choices (observations) that the solver makes against the number of test scenarios that heuristic successfully completes. A good, fast heuristic is one that makes fewer choices (resolving faster) while minimizing restarts.

Our hypothesis was that the main factor in the entropy heuristic's success was that it kept the search space as the frontier of a growing solved continent, rather than an archipelago of islands that might contradict when they grew enough to encounter each other. We expected the heuristics to fall into two broad groups: entropy, lexical, spiral, and hilbert tend to grow contiguous regions, while random and anti-entropy created discontiguous chunks.

As expected, the degenerate case of anti-entropy found the fewest solutions (Fig. 4). While the heuristics that sampled the entire unsolved space were often faster when they reached a solution, there were significantly fewer solutions found. In contrast, the heuristics that focused the search on a growing solved region performed similarly to each other. Surprisingly,

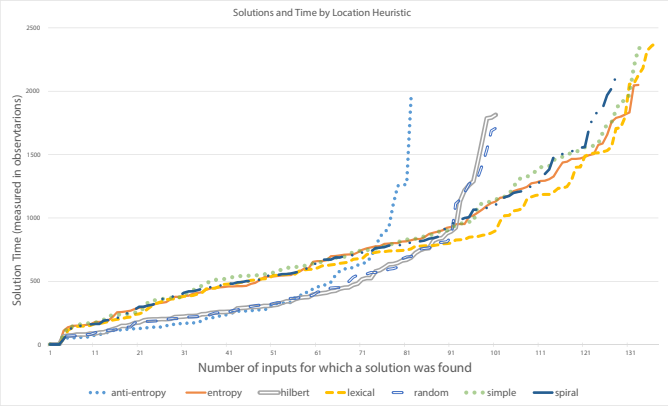


Fig. 4. Cactus plot of location (selection) heuristic experiments. Lines reaching further to the right indicate more test scenarios solved, lower lines indicate that fewer choices (observations) were required. The choice count for a solution include any restarts required because of contradictions. The default limit of 10 restarts was used. There are three clusters of performance: anti-entropy had the fewest solutions, hilbert and random had a few more, and the rest performed comparably. In general, discontiguous location heuristics tend to find solutions slightly faster but are more likely to encounter contradictions.

hilbert performed worse than expected, most likely due to the space-filling curve not interacting well with the dimensions of the output. These findings about which order is best to assign tile codes in a grid roughly confirm the conclusions from a recent paper on transformer-based image generation [25], namely that new tiles should be chosen very close to previously chosen tiles and that a simple lexical ordering (e.g. "row-major" in their terms) was sufficient.

However, when choosing an heuristic, the aesthetic dimension should also be considered. For example, when using a training image with strongly anisotropic details, such as a side view of a city skyline the solutions found tend to be very dependent on the location heuristic used (Fig. 3). Because the

building roofs can only have sky above them, the lexical ordering leads to the roofs being placed earlier than would occur with a random starting location. In this case, lexical ordering strongly biases the generation, reducing the expressive range. While this lexical anisotropy may be a valid aesthetic choice for some purposes, the entropy heuristic performs well on both the performance and aesthetic dimensions, and when in doubt should be the default.

### B. Pattern (Decision) Heuristics Experiment

The second major heuristic is used to decide a value (pattern) for each variable (grid location).

**weighted** The reference implementation uses a random sample weighted by the frequency the pattern appears in the source image.

**random** Uniform random sampling from the domain

**rarest** Choose the least-used pattern; defined as the pattern that appears with the lowest frequency in all the remaining domains

**common** Choose the most-used pattern; defined as the pattern that appears with the highest frequency in all the remaining domains

**lexical** Choose the pattern with the lowest array index

We omit quantitative charts in the interest of space. *Rarest* and *common* frequently run into contradictions. *Lexical* performs surprisingly well, depending on the characteristics of the input, but only *weighted* and *random* find enough solutions to be generally viable. The "local similarity" in Gumin's "locally similar to the input bitmap" includes the "Weak C2" characteristic: [6]

> Distribution of NxN patterns in the input should be similar to the distribution of NxN patterns over a sufficiently large number of outputs. In other words, probability to meet a particular pattern in the output should be close to the density of such patterns in the input.

This is satisfied stochastically by the weighted pattern heuristic: it is likely to sample patterns with the desired characteristic frequency. In practice, many input images are relatively homogeneous and this matters more for scenarios with rare but noticeable patterns.

### C. Backtracking Experiment

A surprising feature that the original WFC algorithm lacked was backtracking. Backtracking is a common feature in constraint solvers [11, Chap. 6], but WFC performed quite well without it, instead opting for a brute-force restarting approach. Some implementations of WFC added backtracking.[19] The question remained: how effective is backtracking for typical WFC problems? There was expressed interest in the wild in adding global constraints.[20] Our hypothesis was that generation with global constraints would be more likely to benefit from the ability to backtrack. To test this, we implemented an "allpatterns" global constraint, which required that every pattern found in the input should be used at least once in the output (Fig. 5).

[19]Including Oskar Stålberg's implementation and the PICO-8 implementation by Rémy Devaux: https://trasevol.dog/2017/09/01/di19/

[20]"No, it's a very good question, I think about global constraints like connectivity a lot. Thanks!" https://twitter.com/ExUtumno/status/760235500858900480

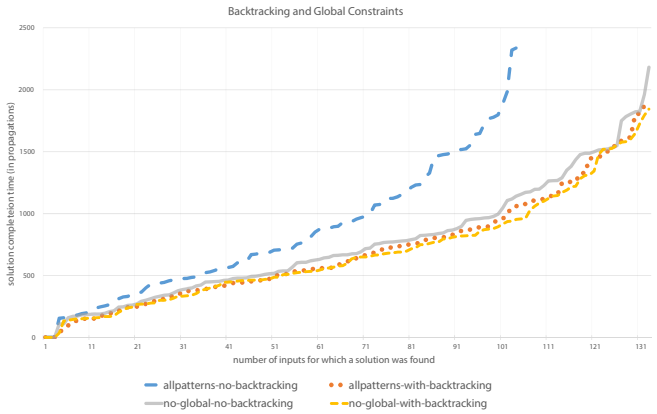


Fig. 5. Cactus plot for backtracking experiments. With only local (adjacency) constraints, backtracking improves performance slightly. Introducing a global constraint without backtracking significantly increases the difficulty (solutions take longer to find and/or fail more often). However, a global constraint with backtracking has comparable performance to non-global-constraint scenarios.

For the test scenario set, backtracking had a minor benefit when only local (adjacency) constraints were used. This is consistent with the general performance of WFC: using constraint solving for procedural generation allows for much more leeway in solution-finding compared to general-purpose constraint solving. With WFC, contradictions are often clashes between deeply incompatible regions in the solution, so shallow backtracking does not have a dramatic effect on the result. In contrast, adding a global constraint has a noticeable impact on performance. Solutions consistently take longer, with significantly more failures. In this case, backtracking makes the global constraint performance comparable to the local constraint performance.[21] This confirms our earlier experimental results with global constraints and backtracking using an ASP-based rational reconstruction of WFC [3].

Importantly, despite the simplicity of adding backtracking to an existing WFC implementation, backtracking makes the search algorithm *complete*: it will always find a solution if one exists, given enough time. Whether a method *may fail* has been a notable feature examined in other example-driven generative methods research [26].

### D. Learned Classification and Rendering Experiment

In most applications of WFC so far, the algorithm was applied to source material built from a small vocabulary of intentionally recombinable tiles. Must these tilesets always be carefully human-curated, or can these be generated as well while still making use of WFC? In this experiment we consider an application where the vocabulary of elementary tiles must be automatically derived from a high-resolution image and the rendering of individual tiles may depend on the context of tiles placed nearby.

As our source image, we used a specific 4096x4096 pixel image of one of the maps from the videogame *WarCraft II:*

[21]This is partially because of the rapid detection of when the global constraint has been violated. Global constraints that require more computation to verify will be correspondingly slower, and global constraints that have more long-range implications will require more backtracking to be successful.

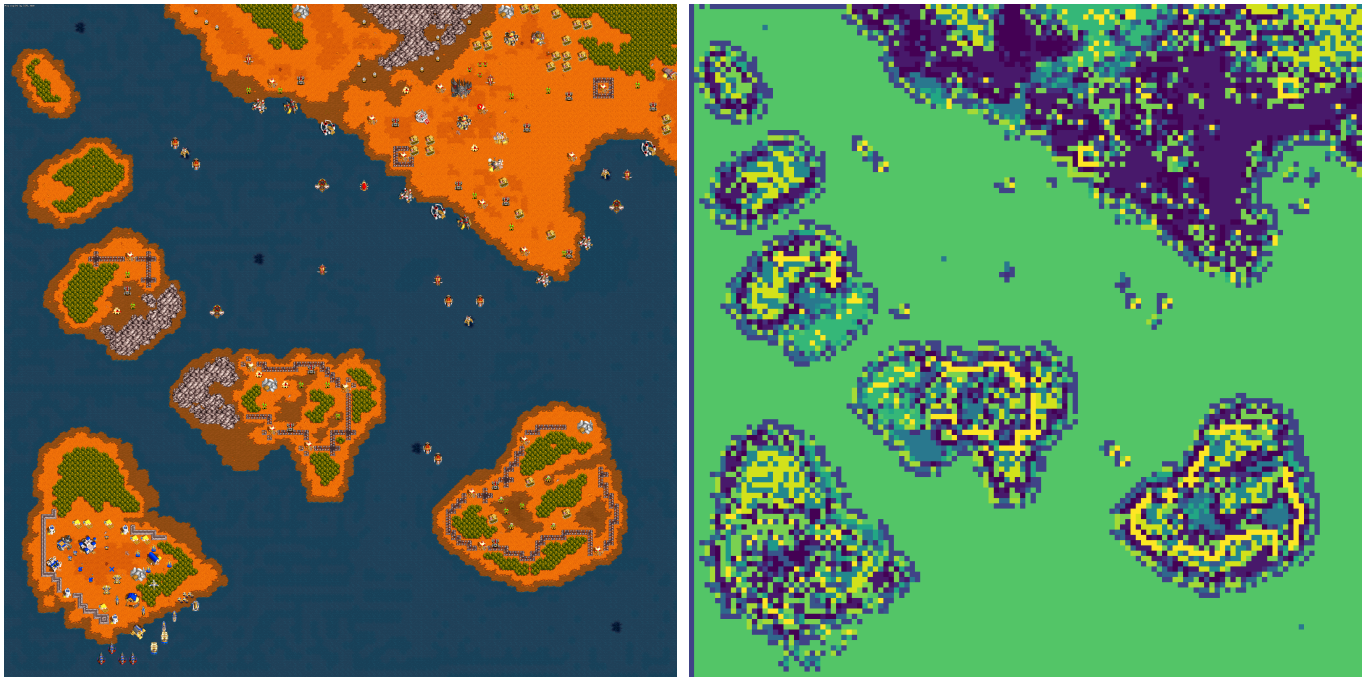

Fig. 6. The left image is the source (training) image for our VQ-VAE experiment (sourced from vgmaps.com). The right image visualizes which of the 16 latent tile codes is assigned to each point on the 128x128-tile grid by our trained VQ-VAE model. Note that the reconstructed appearance of any given latent tile will depend on the tiles placed around it. Zoom in for full detail.

*Tides of Darkness*.[22] We chose this type of image because, although the image is apparently composed of 32x32 pixels tiles, the number of unique tiles needed to exactly reproduce the image is surprisingly high. The map includes several terrain types (water, lowlands, plains, forest, mountain) each with some variations. Between these terrain types are all of the appropriate transitions. For example, the transition from lowlands to water rendered as a shoreline can occur in n/s/e/w as well as diagonal combinations. Additionally, multi-tile buildings and units sit atop the terrain and can cast stippled shadows onto the detail below. Although the overall structure of image is suggestive of a relatively compact data representation, it takes perceptual skill to recover this from the image data given. Figure 6 shows our chosen source image and the tile codes assigned by our learned tile classifier.

Inspired by a line of recent work aiming to bridge continuous neural representations with discrete symbolic representations [27], we implemented a vector-quantized variational autoencoder (VQ-VAE), a kind of neural network that makes use of discrete integer codes in its bottleneck layer. Our network was composed of three subnetworks. The encoder network consumes high-resolution color images and produces a low-resolution grid of high-dimensional vectors (one for each tile on the 32x32 grid). The decoder network consumes grids of that same shape and yields high-resolution color images. Both are shallow, fully-convolutional networks which would together work like a traditional autoencoder with a 512-dimensional bottleneck representation.[23] Between these, we place a vector quantization layer which maps all input vectors to their nearest vector from a trainable codebook with 16 entries. Although each entry in the codebook contains a 512-dimensional vector, it takes just 4 bits of information to identify which codebook entry is used for a given tile. This codebook index, a learned discrete representation for tiles, is the true bottleneck representation in our VQ-VAE.

After training the VQ-VAE on a set of cropped views of the WarCraft source image, we split it into a reusable tile classifier (encoder with attached quantizer) and tile renderer (codebook lookup followed by decoding). Passing the full source image through the learned tile classifier, we get a grid of tile codes. We can then generate novel maps in the style of the specific source image by having WFC operate on local (2x2) patterns of these learned tile codes. Figure 7 shows generation results.

These results demonstrate that WFC does not inherently require a carefully curated tileset. The strategy of using a learned latent tile vocabulary for high-resolution image synthesis here parallels recent work in computer vision (where a transformer-based model replaces WFC for the purposes of generating fresh tile grids with spatial coherence) [25]. Computer vision techniques have previously been used in other work on map generation from images and videos (e.g. in the literature of PCGML [2]), but the focus has been on maps composed of a small vocabulary of identically rendered tiles. In our demonstration here, the value of context-dependent rendering is shown in how local combinations of just 16 latent tile codes can be used to express many more visually-distinct image patches.

[22] Specifically, we used a user-generated image: https://vgmaps.com/Atlas/PC/WarCraftII-TidesOfDarkness-Humans-Mission12-BattleAtCrestfall.png

[23] Additional design details of these networks are beyond the scope of this WFC-focused article.

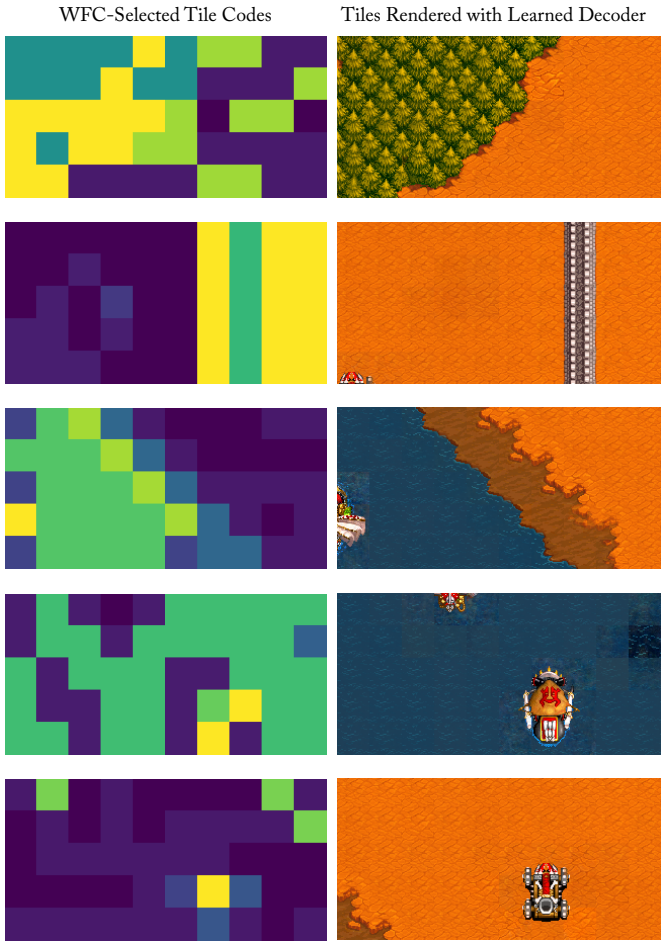


Fig. 7. Novel WFC-generated WarCraft map segments. Images on the left visualize the grid of tile codes (referencing a 512-dimensional vector codebook with 16 entries) output by the solver core of WFC, and those at right show the result of passing those grids through the learned neural decoder. These five samples were chosen from larger batch of 20 unconstrained outputs in order to highlight the variety of terrain transitions and building or unit types.

## V. Tracing WFC in the Wild

### A. Game Development

Within a day of the September 30th 2016 release of WaveFunctionCollapse to the public [28], other developers were actively experimenting with it in the wild. Spreading through social media, particularly via images and animations on Twitter, WFC soon had re-implementations in many environments, including multiple ones for the PICO-8 fantasy console.[24]

One game developer who has contributed to the popularization of WFC is Oskar Stålberg. A technical artist who previously worked on *Tom Clancy's The Division* [29], Stålberg was among the first to start generalizing WFC, extending it with other tile shapes,[25] 3D meshes,[26] performance optimizations,[27] and adding backtracking.[28] In May 2017, as part of a talk about his approach to procedural generation, he released a "small browser demo"[29] to illustrate how his version of the algorithm works under the hood [30]. Notably, Stålberg's WFC started as a clean-room implementation based on the animations rather than on the original code.[30] Stålberg would go on to use his implementation of WFC in the viking invasion game *Bad North*, released in 2018.

One commercially-released indie game that uses WFC is *Caves of Qud* [31], which added partially-WFC-generated villages in July 2018. *Caves of Qud* is a roguelike developed by Freehold Games that is currently in early-access release. One of the developers, Brian Bucklew, started experimenting with using WFC for level generation.[31] *Caves of Qud* uses WFC as part of a pipeline, adding additional pre-generation constraints, using WFC multiple times with different input images, and making further changes to the map after the generation is run.[32] One of the benefits of WFC that *Caves of Qud* has demonstrated is that the simple inputs mean that it is much easier for the entire team to experiment with the generator.[33] The *Caves of Qud* developers noticed two drawbacks of WFC: isotropy and overfitting [32]. While WFC can capture long range correlations, there are limits and large images tend toward isotropy and homogeneity at lower frequencies. Their solution to this was to partition the space with other algorithms and only run WFC on a subset of the level. Likewise, adding more detail to the small training image risked over-constraining the results, as the training overfitted to the image. Their solution was to avoid putting too much detail in the training image and instead add detailing (such as extra doors) using other approaches later in the pipeline.

WaveFunctionCollapse in *Caves of Qud* is an addition to an existing level generator. While *Bad North* also uses WFC as part of a pipeline, the *Bad North* island generator would not exist without WFC. *Bad North* islands are constructed out of 3D modules, designed in Maya and exported into Unity

[24]By Joseph Parker https://twitter.com/jplur_/status/873551783347589120 and TRASEVOL_DOG / Rémy Devaux https://trasevol.dog/2017/06/19/week60

[25]"Content-agnostic algorithm for placing tiles. Heavily based on the work of @ExUtumno. Basically a Sudoku-solver on steroids." https://twitter.com/OskSta/status/784847588893814785

[26]"More procedural tile placement. Now in 3D. Algorithm inspired by the work of @ExUtumno" https://twitter.com/OskSta/status/787319655648100352

[27]"It's getting faster (mostly due to bitwise operations). Actual speed depicted below" https://twitter.com/OskSta/status/794993371261665280

[28]"I gave it an extra difficult tileset to work with to make sure it can repair itself when it has screwed up" https://twitter.com/OskSta/status/793806535898136576

[29]"I built a small browser demo to help explain how the WFC algorithm works. Give it a go: http://oskarstalberg.com/game/wave/wave.html ..." https://twitter.com/OskSta/status/865200072685912064

[30]"It was hard to crack your computer science lingo about collapsing wave functions, so I basically just looked at your gifs" https://twitter.com/OskSta/status/784850280093478912

[31]"The peaceful gardens of Inner Aarranip. A Caves of Qud dungeon generated via https://github.com/mxgmn/WaveFunctionCollapse ... based synthesis." https://twitter.com/unormal/status/805987523596091392

[32]https://forums.somethingawful.com/showthread.php?threadid=3563643&userid=68893&perpage=40&pagenumber=23#post467126402

[33]"Ha! I'm experimenting with the WFC map generator @unormal just added." https://twitter.com/ptychomancer/status/805964921443782656

with a process that calculates which modules have matching geometry along their edges. These "profiles" are used to define the adjacencies: modules with matching profiles are allowed to be adjacent. Some larger modules take up multiple spaces. Variation across islands is achieved by changing the set of modules allowed for an island, thereby creating regions of similar islands by having them use similar sets of tiles and allowing a progression to be created from the input to WFC. The islands are constructed on three-dimensional $11 \times 11$ grids (with some degree of vertical space) and some additional constraints, the most important of which is a connectivity local constraint: the island needs guaranteed paths from the Vikings on the beach to the houses the player is defending.

### B. Artistic

In addition to level design, WFC has been applied to other kinds of content. One of the most unexpected was developed by Martin O'Leary, a glaciologist who also makes "weird internet stuff" [33] including twitter bots and generated travel guides. Inspired by WFC, O'Leary created a poetry generator called Oisín which enforces rhyme/meter constraints to construct sonnets,[34] limericks,[35] and ballads.[36] In personal correspondence with us, O'Leary explained that, "I treat syllables as the basic unit, so each 'tile' is a sequence of syllables (tagged with the word/position it comes from)." This is put into a 1-dimensional WFC sequence, together with "some extra long-distance constraints induced by rhyme, meter, etc." O'Leary has released the source code [24].

Oisín also demonstrates that while WFC is typically used on a rectangular grid, the algorithm has already been extended to graphs. Indeed, Oskar Stålberg demonstrated WFC on the surface of a sphere tessellated with triangles, Github user "Boris the Brave" made an implementation that supports both hexagonal grids and global constraints,[37] and Florian Drux implemented WFC for arbitrary graphs as GraphWaveFunctionCollapse.[38]

### C. Academic

On the research side, WFC has been incorporated into several avenues of research. In "Procedural Content Generation via Machine Learning" (PCGML), Summerville et al. described WFC in relation to Markov random fields and constraint propagation [2].

In "Addressing the Fundamental Tension of PCGML with Discriminative Learning", we use WFC to discuss the tension between having to author enough training data and saving effort. They also discuss how the overlap model's pattern classifier/renderer uses machine learning (similar to autoencoders) and how can use multiple input sources as part of a mixed-initiative design approach [21].

WFC has also been applied to other research problems and generators. Khokhlov, Koh, and Huang developed a voxel synthesis approach based on WFC and Merrell's texture synthesis work, taking advantage of WFC's ability to learn from single-input models [34]. Oliveria et al. used WFC to generate a city as part of an experiment into mixed reality cycling environments [35]. Scurti and Verbrugge used WFC as part of a pipeline to generate example-based paths based on non-programmer input [36]. Finally, Snodgrass includes WFC in a theoretical framework of Markov Models in PCG [37].

## VI. Conclusion

WaveFunctionCollapse is a family of algorithms that have found widespread use across many spheres. The many implementations of WFC vary in details of the pipeline but they all use constraint solving for the propagation of relationships and many use some form of machine learning to learn the constraints from tiny amounts of non-programmer input. In contrast to many parameter-controlled generative methods, WFC starts from a high-level example of the desired output and synthesizes the result. The focus on extremely small amounts of example data has made WFC approachable to many outside of academia in a way that the kinds of data-intensive methods examined the the PCGML literature have not [21].

Our experiments demonstrate the relative importance and resource use of various heuristics and the impact of backtracking in the presence of global design constraints. By examining each facet of the pipeline both individually and in context, we demonstrate paths for future research exploration. In particular, through the connection to vector-quantizing models, we have shown a pathway for integrating recent deep learning techniques while still learning from just one (large) example. Future work might examine the possibility of intentionally learning tile classifiers that lead to compactly expressible (e.g. sparse or low-rank) tile-adjacency matrices or learning rendering functions that take the form of recombinable micro-tiles or other simple representations so that, after training, no neural networks are involved in rendering (allowing the result to be as widely deployed as current WFC variants).

Our explanation of the algorithm is intended to introduce the complex code of the reference implementation in an approachable way. Separating out the different modules in our rational reconstruction enabled us to experiment with heuristics and other aspects of the algorithm's pipeline. Surveying the many implementations and uses for WFC, we have traced how its adoption spread in hobby, academic, and industrial contexts. Following the path of WFC's early adopters, future work should continue to identify and overcome limitations in past variants, demonstrating new ways of using constraint solving and machine learning.

WaveFunctionCollapse presages opportunities for new directions in PCG research. Its influence, already spreading rapidly, ushers us into a new era of PCG that successfully combines techniques that were previously outside the commonly-known PCG sphere.

[34] "Using @ExUtumno's "wavefunction collapse" algorithm to enforce rhyme/meter constraints in text (Alice in Wonderland as Shakespearean sonnet)" https://twitter.com/mewo2/status/789167437518217216

[35] "Also, Pride and Prejudice as a limerick. Turns out the limerick constraints are much harder to satisfy than sonnets, which are easy to write" https://twitter.com/mewo2/status/789177702620114945

[36] "Moby Dick in a conveniently singable ballad form, thanks to WFC" https://twitter.com/mewo2/status/789187174683987968

[37] https://boristhebrave.github.io/DeBroglie/

[38] https://github.com/lamelizard/GraphWaveFunctionCollapse and https://github.com/lamelizard/GraphWaveFunctionCollapse/blob/master/thesis.pdf

## ACKNOWLEDGMENT

The authors would like to thank Oskar Stålberg, Brian Bucklew, Jason Grinblat, Joseph Parker, and Martin O'Leary for their correspondence; and, most importantly, Maxim Gumin for developing WaveFunctionCollapse in the first place.

## REFERENCES

[1] A. M. Smith and M. Mateas, "Answer set programming for procedural content generation: A design space approach," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 187–200, Sept 2011.

[2] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius, "Procedural content generation via machine learning (pcgml)," *IEEE Transactions on Games*, vol. 10, no. 3, pp. 257–270, Sep. 2018.

[3] I. Karth and A. M. Smith, "WaveFunctionCollapse is constraint solving in the wild," in *Proceedings of the 12th International Conference on the Foundations of Digital Games*, ser. FDG '17. New York, NY, USA: ACM, 2017, pp. 68:1–68:10. [Online]. Available: http://doi.acm.org/10.1145/3102071.3110566

[4] A. A. Efros and T. K. Leung, "Texture synthesis by non-parametric sampling," in *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, vol. 2, IEEE. IEEE Computer Society, 1999, 1999, pp. 1033–1038.

[5] L. Liang, C. Liu, Y.-Q. Xu, B. Guo, and H.-Y. Shum, "Real-time texture synthesis by patch-based sampling," *ACM Transactions on Graphics (ToG)*, vol. 20, no. 3, pp. 127–150, 2001.

[6] M. Gumin. (2017, May) WaveFunctionCollapse Readme.md. [Online]. Available: https://github.com/mxgmn/WaveFunctionCollapse/blob/master/README.md

[7] P. C. Merrell, "Model synthesis," Ph.D. dissertation, University of North Carolina at Chapel Hill, 2009.

[8] M. Gumin, "Syntex," *GitHub repository*, 2016. [Online]. Available: https://github.com/mxgmn/SynTex

[9] ——, "Convchain," *GitHub repository*, 2016. [Online]. Available: https://github.com/mxgmn/ConvChain

[10] L. A. Gatys, A. S. Ecker, and M. Bethge, "A neural algorithm of artistic style," *CoRR*, vol. abs/1508.06576, 2015. [Online]. Available: http://arxiv.org/abs/1508.06576

[11] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Pearson Education, 2016.

[12] L. Foged and I. Horswill, *Rolling Your Own Finite-Domain Constraint Solver*. A K Peters/CRC Press, 2015, pp. 283–302.

[13] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[14] I. D. Horswill and L. Foged, "Fast procedural level population with playability constraints," in *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[15] G. Smith, J. Whitehead, and M. Mateas, "Tanagra: Reactive planning and constraint solving for mixed-initiative level design," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 201–215, 2011.

[16] C. Prud'homme, J.-G. Fages, and X. Lorca, *Choco Documentation*, TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016. [Online]. Available: http://www.choco-solver.org

[17] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, *Answer Set Solving in Practice*, ser. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, 2012.

[18] M. Gebser, B. Kaufmann, and T. Schaub, "Conflict-driven answer set solving: From theory to practice," *Artificial Intelligence*, vol. 187, pp. 52–89, 2012.

[19] R. Kaminski, T. Schaub, and P. Wanko. (2017) A tutorial on hybrid answer set solving with clingo. [Online]. Available: https://www.cs.uni-potsdam.de/~torsten/hybris.pdf

[20] M. Gumin, "Wavefunctioncollapse," *GitHub repository*, 2016. [Online]. Available: https://github.com/mxgmn/WaveFunctionCollapse

[21] I. Karth and A. M. Smith, "Addressing the fundamental tension of pcgml with discriminative learning," in *Proceedings of the 14th International Conference on the Foundations of Digital Games*, ser. FDG '19. New York, NY, USA: ACM, 2019, pp. 89:1–89:9. [Online]. Available: http://doi.acm.org/10.1145/3337722.3341845

[22] M. Fehr and N. Courant, "fast-wfc," *GitHub repository*, 2018. [Online]. Available: https://github.com/math-fehr/fast-wfc

[23] C. P. Gomes, A. Sabharwal, and B. Selman, "Near-uniform sampling of combinatorial spaces using xor constraints," in *Advances in Neural Information Processing Systems*, 2006, pp. 481–488.

[24] M. O'Leary. (2017, May) Oisín: Wave function collapse for poetry. [Online]. Available: https://github.com/mewo2/oisin

[25] P. Esser, R. Rombach, and B. Ommer, "Taming transformers for high-resolution image synthesis," 2020.

[26] S. M. Lucas and V. Volz, "Tile pattern kl-divergence for analysing and evolving game levels," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2019, pp. 170–178.

[27] A. van den Oord, O. Vinyals, and K. Kavukcuoglu, "Neural discrete representation learning," 2018.

[28] M. Gumin. (2016, Sep) Bitmap & tilemap generation from a single example by collapsing a wave function https://github.com/mxgmn/wavefunctioncollapse. [Online]. Available: https://twitter.com/ExUtumno/status/781834584136814593

[29] T. Holmes. (2016, Jan) Interview with phenomenal game designer oskar stälberg. [Online]. Available: https://taylorholmes.com/2016/01/22/interview-with-phenomenal-game-designer-oskar-stalberg/

[30] O. Stålberg. (2017, May) wave.html. [Online]. Available: http://oskarstalberg.com/game/wave/wave.html

[31] J. Grinblat and C. B. Bucklew, "Caves of Qud," 2010.

[32] C. B. Bucklew, "Roguelike celebration 2019," Oct 2019. [Online]. Available: https://www.youtube.com/watch?v=fnFj3dOKcIQ

[33] M. O'Leary. (2017) Twitter bio. [Online]. Available: https://twitter.com/mewo2

[34] M. Khokhlov, I. Koh, and J. Huang, "Voxel synthesis for generative design," in *Design Computing and Cognition '18*, J. S. Gero, Ed. Cham: Springer International Publishing, 2019, pp. 227–244.

[35] W. Oliveira, W. Gaisbauer, M. Tizuka, E. Clua, and H. Hlavacs, "Virtual and real body experience comparison using mixed reality cycling environment," in *Entertainment Computing – ICEC 2018*, E. Clua, L. Roque, A. Lugmayr, and P. Tuomi, Eds. Cham: Springer International Publishing, 2018, pp. 52–63.

[36] H. Scurti and C. Verbrugge, "Generating paths with wfc," in *Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2018.

[37] S. Snodgrass, "Markov models for procedural content generation," Ph.D. dissertation, Drexel University, 2018.