WaveFunctionCollapse is Constraint Solving

Anonymous Author(s)

ABSTRACT

Maxim Gumin's WaveFunctionCollapse (WFC) algorithm is an example-driven image generation algorithm emerging from the craft practice of procedural content generation. In WFC, new images are generated in the style of given examples by ensuring every local window of the output occurs somewhere in the input. Operationally, WFC implements a non-backtracking, greedy search method. This paper examines WFC as an instance of constraint solving methods. We trace WFC's explosive influence on the technical artist community, explain its operation in terms of ideas from the constraint solving literature, and probe its strengths by means of a surrogate implementation using answer set programming.

ACM Reference format:
Anonymous Author(s). 2017. WaveFunctionCollapse is Constraint Solving. In *Proceedings of 8th Procedural Generation Workshop, International Conference on the Foundations of Digital Games 2017, PCG, Aug 2017 (FDG17),* 9 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Constraint solving is neither a traditional nor well-known approach to procedural content generation (PCG). Nevertheless, this approach can be surprisingly effective for building controllable content generators. This paper examines the WaveFunctionCollapse (WFC) algorithm, an example-driven image generation algorithm recently developed by game developer Maxim Gumin, and illuminates it through the lens of constraint solving.

In 2016 and 2017, WaveFunctionCollapse attracted the attention of several indie and hobby game makers via social media.[1] Animations of the algorithm demonstrated not just the primary output, a large image generated in the style of a smaller artist-provided source, but also a surprisingly human-like mode of incremental creation. Referring to one of the major components of the algorithm, Gumin writes [12]: "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." The animated visualizations, with the results gradually resolving themselves out of a fog of possibilities (Fig. 1), instantly showed that the algorithm works differently than familiar constructive or artifact-at-a-time generate and test methods.

[1] https://twitter.com/ExUtumno/status/781833475884277760
https://twitter.com/dwtw/status/810166761270243328
https://twitter.com/jplur_/status/784591710777147392

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FDG17, PCG
© 2017 ACM. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

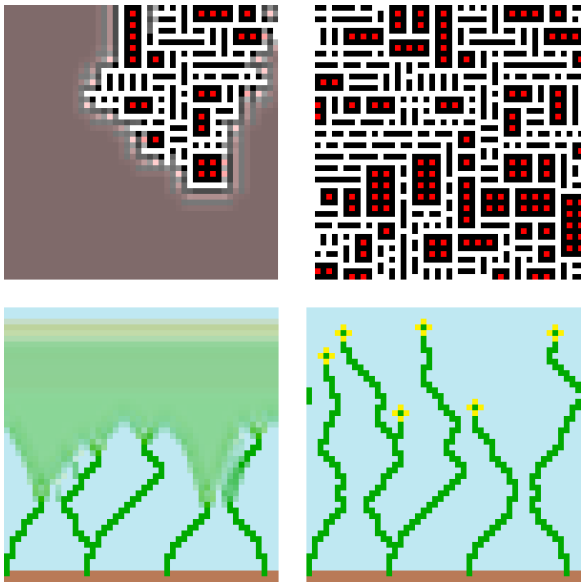

Figure 1: The WaveFunctionCollapse generator in action. The unresolved information in the images on the left is shown as the average color value of their possible outputs.

The approach used draws deeply from techniques known in computer graphics for texture synthesis (we examine this literature more in a later section). Where WaveFunctionCollapse departs from texture synthesis is also a key place where it enables surprising new applications in game design—it does not allow pixel colors to be blended, preserving gameplay semantics demonstrated in the source image. In this sense, it is similar to generative methods based on Markov chains that, in applications like generative text (e.g. Michael Walker's King James Programming [27]), assemble outputs from locally-consistent chains of unmodified text. Both traditional texture synthesis and Markov chain approaches are primarily data-driven and thus accessible to non-programmers, a feature that stands in contrast with expectations about systems that operate on the basis of constraint solving.

In this paper, we make the following contributions:

- We trace significant applications of WaveFunctionCollapse to date.
- We provide a pseudocode explanation of the algorithm's major steps in the context of a worked example and the terminology of constraint solving.
- We reformulate the image generation problem using the technology of answer set programming to interrogate the source of the algorithms strengths in relation to traditional constraint solving techniques.

This work aims to highlight the value of studying the craft practice of PCG and to clarify the present and future role of constraint

solving as one of the many generative methods we can draw on in applied PCG systems.

2 BACKGROUND

In this section, we relate the WaveFunctionCollapse algorithm to some of the earlier approaches to image generation and introduce the vocabulary that will help us examine WFC as a constraint solving algorithm.

2.1 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 [1]. In many texture synthesis approaches (e.g. the work of Liang et al. [17]), 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). Although different algorithms achieve this goal in different ways, many texture synthesis algorithms 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 WaveFunctionCollapse (towards important implications for game development discussed later).

In Liang's method, [17] 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. WaveFunctionCollapse 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, WFC needs to perform many more bookkeeping operations in the not-yet-known regions of the output in response to the need for exact pattern matching.

While WFC is loosely inspired by quantum mechanics,[2] Gumin writes that he was inspired by the discrete synthesis approach of Paul Merrell [18]. Although Merrell worked in computer graphics and was also inspired by texture synthesis, he focused on the problem of generating three-dimensional geometric models. In this setting, we want to automatically 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 geometric chunks.

In personal correspondence with us, Gumin described how he was inspired by convolutional neural network style transfer, but found it lacking for 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 pattern in the input. Gumin's intent was to capture the rules for how the source image was made.

His SynTex project [10] implemented several texture synthesis methods, yielding attractive results for game texture images but nonsensical outputs for non-texture images (such as of items like swords) where pixel-grid analysis destroyed the visual semantics of structured objects. In the ConvChain project [9], he experimented with an approach based on Markov Chain Monte Carlo, 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 Gumin's later WaveFunctionCollapse algorithm.

2.2 Constraint solving algorithms

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 [22, 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 WaveFunctionCollapse, 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 influence 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.

[2] Very loosely, and mostly confined to how it uses 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." [12]

[3] Recent innovations in style transfer were sparked by a breakthrough in using deep convolutional neural network classifiers to mimic artistic styles [4] This has lead to a flood of related research, along with the exploration of other applications of neural networks to graphics.

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 selection 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 eventually, in finite time, return a solution 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. AC3 is a well known constraint propagation algorithm [22, Chap. 6]. Although AC3 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" [2] for more details.

2.3 Constraint solving in PCG

Although there are a few examples of note, constraint solving is mostly overlooked for the purposes of content generation. Taxonomies of PCG such as in the notable search-based PCG survey [26] 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 WaveFunctionCollapse executions so visually stunning—we aren't 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 (AC3) 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.

In G. Smith's Tanagra system, [24] a mixed-initiative platformer level design tool, the Choco [21] 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 contrast, Gumin's WaveFunctionCollapse does not backtrack.

2.4 ASP in PCG

Answer set programming (ASP) is a form of logic programming targeted at modeling combinatorial search and optimization problems [5]. 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 [6].

A. M. Smith proposed the use of ASP in PCG [23] within the paradigm of modeling design spaces. Rather than directly aiming to code and algorithm for generating content, we should declaratively model the space of content we want to see and let a domain-independent solver take care of the procedural aspects for us. 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. That background is not common amongst those, predominantly technical artists, who were recently excited find WaveFunctionCollapse.

Modern answer set solvers (such as Clingo [15]) 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. We will make use of Clingo later in this paper to implement an experimental surrogate for WaveFunctionCollapse on top of advanced constraint solving algorithms to better understand the features of Gumin's invention.

3 WFC IN THE WILD

Within a day of the September 30th 2016 release of WaveFunctionCollapse to the public [8], other developers were actively experimenting with it in the wild.

Joseph Parker, an indie game developer, stated on Twitter[4] that he had, "never been this excited about an algorithm!" Parker immediately started work on a toolset for the Unity3D engine,[5] releasing it as a Unity asset before the end of October.[6] This toolset was, in turn, quickly in active use by others.[7]

An active member of the experimental procedural generation community, Parker had previously participated in ProcJam 2015. ProcJam 2016 was the next month, and Parker's entry was *Proc Skater 2016*, developed along with Ryan Jones and Oscar Morante. [20] *Proc Skater 2016* was the first game to use WaveFunctionCollapse for level generation [8], generating skate parks from the designer's sampled input. The output skate parts were formed by arranging discrete geometric chunks (akin to Merrell's discrete model synthesis) for which exact matching of local patterns ensured the smooth traversability required during gameplay. Fig. 2 shows a screenshot of the level generation.

[4] https://twitter.com/jplur_/status/784591710777147392
[5] https://twitter.com/jplur_/status/782271940694306816
[6] https://twitter.com/jplur_/status/792440594845032448
[7] https://twitter.com/oh_cripes/status/807565996957564928
[8] https://twitter.com/ExUtumno/status/812703329834962944

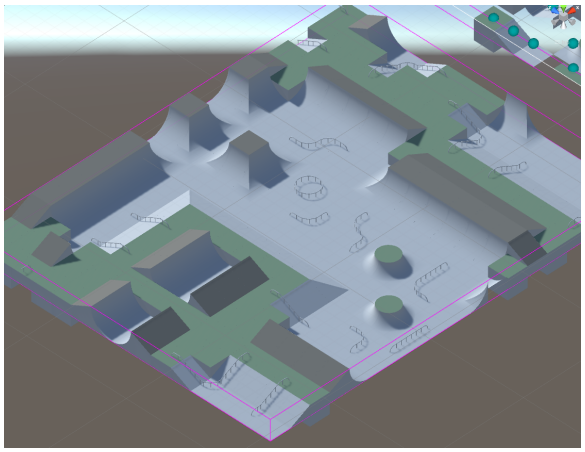

Figure 2: Behind the scenes of level generation in the first use of WaveFunctionCollapse in a game, *Proc Skater 2016*. [20] Copyright 2016 Joseph Parker.

Another game developer who has contributed to the popularization of WaveFunctionCollapse is Oskar Stålberg. A technical artist who previously worked on *Tom Clancy's The Division*, [13] Stålberg was among the first to start generalizing WaveFunctionCollapse, extending it with other tile shapes,[9] 3D, meshes[10] performance optimizations,[11] and adding backtracking.[12] In May 2017, as part of a talk about his approach to procedural generation, he released a "small browser demo"[13] to illustrate how the algorithm works under the hood [25]

WaveFunctionCollapse has also been used in commercially-released indie games, most notably *Caves of Qud* [3]. *Caves of Qud* is a roguelike developed by Freehold Games that is currently in early-access release. Brian Bucklew, one of the developers, started experimenting with using WaveFunctionCollapse for level generation.[14] Two of the levels are shown in Fig. 3). *Caves of Qud* uses a multipass WFC system, with templates applied successively to combine into a larger variety of outcomes with more extreme variation.[15] 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.[16]

In addition to level design, WaveFunctionCollapse 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" [19] including twitter bots and procedurally generated travel guides. O'Leary created a poetry generator inspired by WaveFunctionCollapse that enforced rhyme/meter constraints to make sonnets from Alice in Wonderland,[17] Pride and Prejudice

[9]https://twitter.com/OskSta/status/784847588893814785
[10]https://twitter.com/OskSta/status/787319655648100352
[11]https://twitter.com/OskSta/status/794993371261665280
[12]https://twitter.com/OskSta/status/793806535898136576
[13]https://twitter.com/OskSta/status/865200072685912064
[14]https://twitter.com/unormal/status/805987523596091392
[15]https://forums.somethingawful.com/showthread.php?threadid=3563643&userid=68893&perpage=40&pagenumber=23#post467126402
[16]https://twitter.com/ptychomancer/status/805964921443782656
[17]https://twitter.com/mewo2/status/789167437518217216

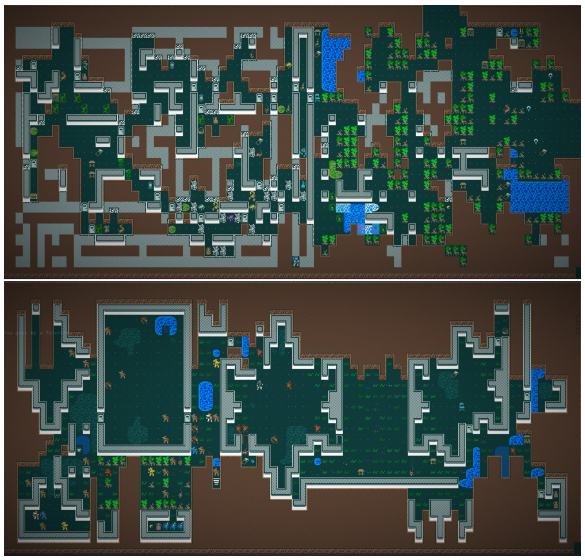

Figure 3: Two different historical site levels in *Caves of Qud* generated via WaveFunctionCollapse. Copyright 2016 Freehold Games.

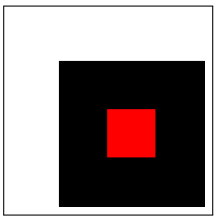

Figure 4: The 4×4 pixel Red Maze sample, used as a source image. Note that it tiles periodically, with the edges wrapping around.

as a limerick,[18] and "Moby Dick in a conveniently singable ballad form."[19] 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 entered into a 1-dimensional WFC sequence, together with "some extra long-distance constraints induced by rhyme, meter, etc."

In this setting, the texture synthesis view of WFC (operating on images composed of pixels) is not nearly as informative as a constraint solving view where the algorithm is seen to make choices for variables from domains in a way that avoids violating stated constraints.

4 THE WFC ALGORITHM

In this section, we examine the details of Gumin's original formulation of the WaveFunctionCollapse algorithm [11]. 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

[18]https://twitter.com/mewo2/status/789177702620114945
[19]https://twitter.com/mewo2/status/789187174683987968

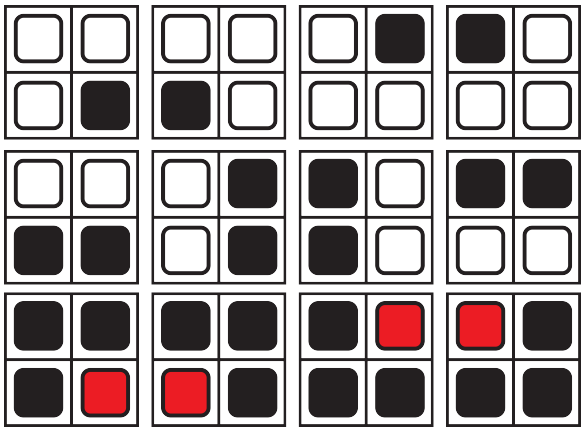

Figure 5: The patterns derived from the Red Maze sample, with a pattern size of $N = 2$, reflection, and rotation. Their order here, from left to right, top to bottom, is the same as the index ordering used in the original C# implementation. Note that only patterns that appear in the original sample are present.

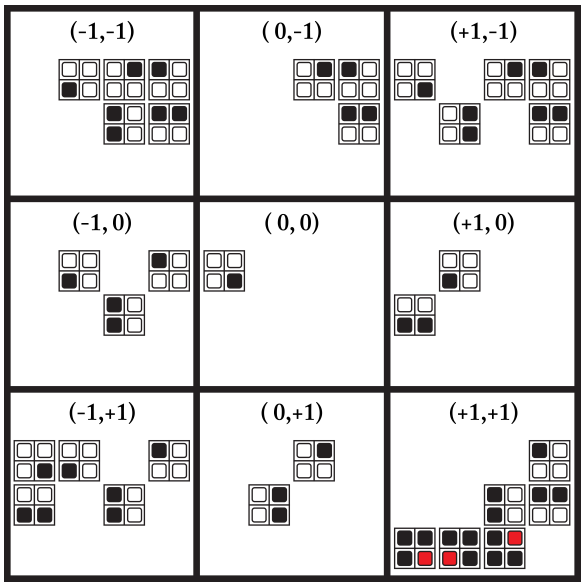

Figure 6: A slice of the index datastructure for the first pattern in Red Maze, showing which of the other patterns can overlap with it. Note that, of course, the only pattern that can overlap with it at zero offset is itself. The index datastructure also stores similar information for the other patterns.

the algorithm are difficult to interpret by reading the code directly. In personal correspondence with several users of WFC, we learned that many of them treated the code as a black box, using it directly without attempting to alter it. In response, we offer a pseudocode summary below.

Throughout the explanation of the algorithm, we'll use one of Gumin's sample files as a running example. The `Red Maze.png` image (Fig. 4) is a compact, three-color sample that can produce a wide variety of outcomes, showing the features of the generator. At the same time, the 16 source pixels and limited number of patterns makes it easy to follow.

At the top level, WFC performs four key tasks: it extracts the local patterns from the input image; it processes those patterns into an index that speeds up constraint checking; it incrementally generates the output image by growing a partial assignment; and it finally renders the total assignment back into an image in the same format as the input.

```
defn Run():
    PatternsFromSample()
    BuildPropagator()
    Loop until finished:
        Observe()
        Propagate()
    OutputObservations()
```

A *pattern* here is a particular, unique configuration of input tiles. In the simple tiled version of the algorithm, the patterns are specified as explicit tile constraint relationships. In the overlapping tile version, the constraints are inferred from the source image, constructing a set of the unique $N \times N$ patterns from subimages (Fig. 5). Symmetry and reflection can optionally be taken into account.

As can be seen in Fig. 5, when $N = 2$, the maze sample contains twelve unique patterns. Four with a single black pixel, four with two black and two white, and four around the red pixel. The red and white pixels are never next to each other in the source image, so there is no pattern with that combination. Note that the sampled image is periodically tiling. This is optional, and only relevant for cataloging the patterns. But makes it much easier to specify some classes of input.

From the set of patterns, `BuildPropagator()` creates an index datastructure that describes the ways that the patterns can be placed near one another. For the overlap version, the index contains the pre-calculated answers to whether the union between two patterns match when one placed near the other with a particular x,y offset. (When $N = 3$ there are $(2(N-1)+1)^2 = 36$ offsets to consider.) For the tiled version, this index can be created directly from the designer-specified tile relationships. In either case, this creates a sparse relation between the patterns (sparse in the sense that most patterns cannot occur with most offsets to most other patterns). Although Gumin's code refers to this index as `propagator` in C#, we here call it an index to avoid confusion with (constraint propagation) propagators like AC3 (which WFC implicitly implements).

During the core incremental generation process, decision variables (grid locations) are repeatedly selected and then assigned. In constraint solving, in addition to the current partial assignment, solvers typically keep track of remaining domains for unassigned variables. In Gumin's C# code, this is stored in a table called `wave` in loose reference to a quantum wave function. The entries of the table, which Gumin calls the coefficients, are Boolean values that record whether or not the algorithm might yet still assign a given

pattern to a given location. All coefficients in the wave are initialized to a true value, which is equivalent to saying each decision variable has an unreduced initial domain. Assignment and propagation both serve to pare down the domains of variables. Accordingly, coefficients only go from true to false during the execution of WFC. Gumin's algorithm does not implement local backtracking and instead globally restarts in the rare case a conflict is reached.

```
defn Observe(coefficient_matrix):
   FindLowestEntropy()
   If there is a contradiction, throw an error and quit
   If all cells are at entropy 0, processing is complete:
      Return CollapsedObservations()
   Else:
      Choose a pattern by a random sample, weighted by the
           pattern frequency in the source data
      Set the boolean array in this cell to false, except
           for the chosen pattern
```

The purpose of `Observe()` is to identify the location on the grid with the lowest entropy nonzero. Entropy here corresponds to the interpretation of the wave as implying a probability distribution over the patterns to be found at each grid location. The cell with lowest entropy is the variable with the tightest or smallest domain after propagation. The heuristic of selecting the *most constrained variable* or equivalently the variable with *minimum remaining values* (MRV) is well known in constraint solving [22, Chap. 6].

```
defn FindLowestEntropy(coefficient_matrix):
   Return the cell that has the lowest greater-than-zero
        entropy, defined as:
      A cell with one valid pattern has 0 entropy
      A cell with no valid patterns is a contradiction
      Else: the entropy is based on the sum of the frequency
           that the patterns appear in the source data, plus
           Use some random noise to break ties and
                near-ties.
```

Since there is more than one valid pattern for that location—or it would already have been set to zero entropy in the previous loop—one of those patterns needs to be chosen. One of the patterns is chosen with a random sample, weighted by the frequency that pattern appears in the input 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 [12].

Once a location has been observed (a variable has been assigned), it is flagged as a location in the wave to be updated (as a place to start updating variable domains via constraint propagation). Like AC3, 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. Updating the domain of one variable implies the need to potentially update all of the adjacent variables. As such, propagation proceeds via an algorithm recognizable from a graphics perspective as a flood fill.

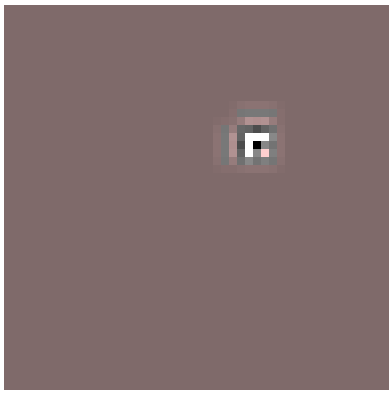

Figure 7: The result of the first observation and propagation step with the Red Maze sample. Since all of the locations have equal entropy, the start was chosen at random. The selected pattern is the the first one, with a single black pixel in the lower right corner. Note that the propagation has already resolved two additional white pixels, since every remaining pattern that can cover those locations has a white pixel in that location.

```
defn Propagate(coefficient_matrix):
   Loop until no more cells are left to be update:
      For each neighboring cell:
         For each pattern that is still potentially valid:
            Compare this location in the pattern with the
                 cell's values
               If this point in the pattern no longer
                    matches:
                  Set the array in the wave to false for this
                       pattern
                  Flag this cell as needing to be updated in
                       the next iteration
```

Each observation finalizes the result of one location, and reduces the entropy of the surrounding region (Fig. 8).

Once there is no more entropy in the system (all variables have a singleton domain), we can output the final generated image (Fig. 9). Additionally, we can take advantage of the side-effect of each cell having an array of potential states and output a partially-finished image after each cycle of observation and propagation. This is what allowed the enticing visualizations noted in Fig. 8.

```
defn OutputObservations(coefficient_matrix):
   For each cell:
      Set observed value to the average of the color value
           of this cell in the pattern for the remaining
           valid patterns
   Return the observed values as an output image
```

Taken together, we can see WaveFunctionCollapse as a constraint solving algorithm. Indeed, Gumin occasionally describes his algorithm this way.[20] It uses the minimum remaining values (MRV) heuristic to select a variable to decide next. For decisions, it uses

[20] https://twitter.com/ExUtumno/status/793601984800624640

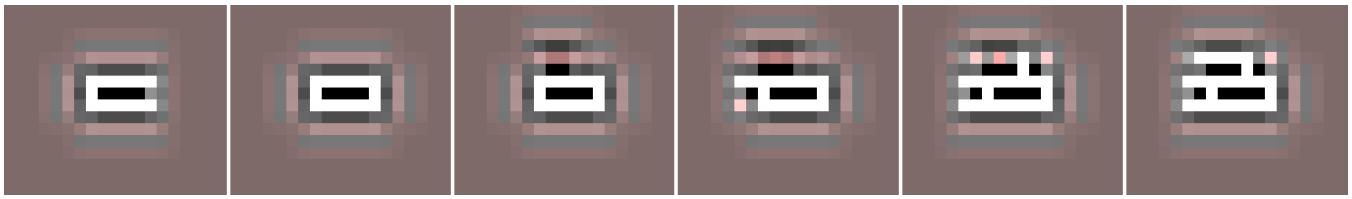

Figure 8: Consecutive updates with the Red Maze dataset. Partially resolved cells are rendered as the average of their potential outputs. Note how the propagation spreads to more cells in some steps, as the falling entropy allows multiple cells to be resolved in the same step.

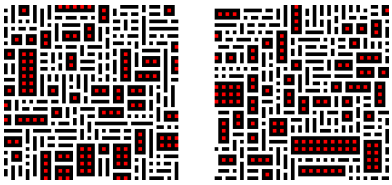

Figure 9: Two of the possible outcomes of generation using the Red Maze sample.

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 [22]. 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 [7].

5 REFORMULATING WFC IN ASP

In this section, we use answer set programming (ASP) to implement a surrogate for WaveFunctionCollapse. It is not a reimplementation of WFC *per se* (for example, we do not attempt to capture Gumin's entropy heuristic) but instead an attempt to capture the problem WFC solves in order to support asking questions about how WFC might have been implemented differently.

Our surrogate reuses Gumin's original input data examples and input processing algorithms. Just before Gumin's observe-and-propagate cycle begins, we extract the index of legal pattern adjacencies (what Gumin calls the `propagator`) as well as the topology of the grid that the algorithm is about to fill.

Our formulation of the image generation problem in ASP involves just two rules:

```
1 { assign(X,Y,P):pattern(P) } 1 :- cell(X,Y).

:- adj(X1,Y1,X2,Y2,DX,DY),
   assign(X1,Y1,P1),
   not 1 { assign(X2,Y2,P2):legal(DX,DY,P1,P2) }.
```

The first rule states that every cell should be nondeterministically assigned exactly one pattern. The next rule is an integrity constraint (which disallows certain solutions). It states that a solution should be rejected if there is an adjacency between two cells of a certain spatial offset and the first cell is assigned one pattern and the second cell is not assigned one of the patterns marked as legal according to the index.

Combinging these two rules with a set of instance-specific facts automatically derived from the snapshot mentioned above, we have a logic program that can be solved with Clingo (we use Clingo 5.2.0[21]).

When Clingo runs, two major tasks occur. First, the logic program is grounded: symbols from the problem instance are substituted for all logic variables in the problem formulation. This yields a low-level constraint problem. The time taken during grounding for this problem formulation is proportional to the number of grid cells multiplied by the number of legal pairings of patterns mentioned in the index. Rather than checking all possible offsets between patterns, we only consider those in the four cardinal directions on the grid (such that $|DX| + |DY| = 1$) because this constraint subsumes the longer-range constraints. It bears noting that this formulation does not pay the cost of all possible combinations of local pixel values nor even all possible pairs of patterns present in the input image. Like Gumin's WFC, we take care to only do work proportional to the number of sparse pairings of patterns.

After grounding, the generated constraint problem is solved to find one or more satisfying assignments. By adjusting parameters of the solver, we can cause Clingo to imitate various traditional search algorithms. In the case where a solution can be found without backtracking, solving takes time proportional to a modest polynomial in the number of decision variables and constraints in the CSP (where the details depend on the precise datastructure design choices used in constraint propagation). When backtracking does occur, the time taken is difficult to characterize beyond that it is related to the number of dead-ends (conflicts) encountered during search. In the discussion below, we focus on the conflict counts rather than wall-clock times to factor out the performance of the specific machine used for testing.[22]

To focus our experiments, we selected three of Gumin's scenarios, illustrated in Fig. 10: `Flowers`, `Platformer`, and `Skyline` (each with $N = 3$).

5.1 Understanding Heuristics

In our first experiment, we aim to understand the importance of Gumin's entropy heuristic. We do this first asking Clingo to run with all of the built-in heuristics disabled (passing the "`--heu=none`" command line flag). This has the effect of causing Clingo to select

[21] https://github.com/potassco/clingo/releases/tag/v5.2.0

[22] For reference, all non-timeout solving times were under two seconds using single threaded search on a Early 2011 MacBookPro with a 2.2 GHz Intel Core i7 processor

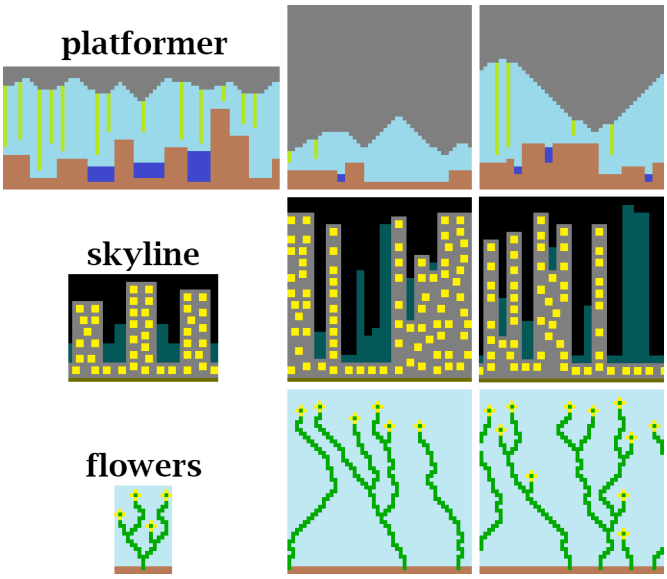

Figure 10: Three input images on the left, paired with their example outcomes on the right.

grid location in some default ordering (namely the reading-order traversal of the grid we used when preparing the problem instance facts) and to choose patterns for those cells by taking the first not-eliminated pattern from the domain.

Surprisingly, Clingo encounters zero conflicts during search for the selected scenarios. This result still holds if we tell Clingo to make random choices for each selected location (something needed to achieve varied outputs for gameplay purposes). This suggests that the strength of WFC comes from constraint propagation (removing bad choices from variable domains before they are considered for assignment) rather than the entropy heuristic. Both the entropy heuristic and the effect of Clingo's disabled heuristics have similar behavior: the next cell selected to be assigned is often next to cells that have already been assigned.

Clingo comes with a domain-agnostic heuristic known as VSIDS (Variable State Independent Decaying Sum [16]), a dynamic heuristic that learns how to make good choices at run-time by observing where past choices failed. If we give VSIDS a chance to solve the image generation problem, we find it does nearly as well as above: just a few conflicts are encountered. Again, this suggests the importance of constraint propagation over heuristics.

Interestingly, a baseline heuristic that decides which cell to assign next randomly represents a pathological choice for this search task. This process often selects cells that are not near any other assigned cell, inviting the opportunity for many choices that need to be backtracked later. Given a minute to search, the solver always times out (after many thousands of conflicts) when making random selections for which cell to assign next.

5.2 Understanding Backtracking

The results above would suggest that (when using non-pathological heuristics) backtracking is not important. From this, it makes sense that Gumin was able to achieve reasonable results by simply globally restarting in the greedy search if a conflict is encountered.

By adding some reasonable global constraints to our ASP formulation, we can probe how brittle this result is. Continuing with the spirit of the image generation task, we consider adding the following integrity constraint to our formulation above:

```
:- pattern(P), not 1 { assign(X,Y,P):cell(X,Y) }.
```

This constraint says to reject a possible solution if there is some pattern (from the input image) that isn't used in at least one assignment for the output image.

Experimentally, we found that while adding this constraint did not significantly impact the number of conflicts encountered for the Flowers and Platformer scenarios, it leads to hundreds of conflicts for the Skyline scenarios. If the the Clingo is instructed to globally restart after each conflict, it cannot find a solution within the one-minute timeout window. However, if local backtracking is allowed (the default behavior of Clingo), the constraint can be quickly resolved by adjusting local choices.

In deeper game design applications of WaveFunctionCollapse that attribute gameplay semantics to what are just pixel colors in the image generation task, we expect the demand for global constraints like this to grow. For example, consider an application that attempts to use WFC to generate an explorable environment.[23] It seems desirable to be able to ask the search algorithm to enforce global reachability constraints: every location which the player might occupy should have a feasible path from the initial location in the environment. A designer might specify this by identifying a certain pixel color in the input image and flagging that color as needing to form a single connected graph (a global constraint). A balance of local backtracking and global restarts [28] will be needed in the search algorithm to efficiently generate designs satisfying this constraint.

6 CONCLUSION

We have shown that WaveFunctionCollapse is a significant application of constraint solving for PCG with multiple in-the-wild uses. Because WFC works with abstract chunks of content rather than literal, blendable color values, it has many exciting applications such as poetry and constrained level generation. Through experiments with the ASP surrogate implementation, we show that WFC's choice of heuristic and decision to only apply global restarts of search are reasonable choices for the original discrete image generation task, but they are not critical going forward. Indeed, local backtracking is being added to WFC by others such as Oskar Stålberg who are reconsidering some of the Gumin's original algorithm and datastructure design choices. We assert that search in the space of partial assignments and constraint propagation are the primary strengths of WFC.

As a data-driven content generator with performance attractive to many practitioners, WaveFunctionCollapse serves to upend many prior expectations about the properties of constraint solving methods in PCG. As we can see from the enthusiastic uptake

[23] such as the roguelike dungeons in *Caves of Qud* or the 3D environments in Oskar Stålberg's experiments: https://twitter.com/OskSta/status/797119718477991936

of the algorithm by artists and designers, the data-driven content generation is more accessible. Even though many users treat the algorithm as a black box, they are able to effectively use it to create interesting content.

REFERENCES

[1] Alexei A Efros and Thomas K Leung. 1999. Texture synthesis by non-parametric sampling. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, Vol. 2. IEEE, 1033–1038.

[2] Leif Foged and Ian D Horswill. 2015. *Rolling Your Own Finite-Domain Constraint Solver.* A K Peters/CRC Press, 283–302.

[3] Freehold Games. 2017. Caves of Qud. (2017).

[4] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. 2015. A Neural Algorithm of Artistic Style. *CoRR* abs/1508.06576 (2015). http://arxiv.org/abs/1508.06576

[5] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2012. *Answer Set Solving in Practice.* Morgan and Claypool Publishers.

[6] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. 2012. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence* 187 (2012), 52–89.

[7] Carla P Gomes, Ashish Sabharwal, and Bart Selman. 2006. Near-Uniform Sampling of Combinatorial Spaces Using XOR Constraints. In *Advances in Neural Information Processing Systems*. 481–488.

[8] Maxim Gumin. 2016. Bitmap & tilemap generation from a single example by collapsing a wave function https://github.com/mxgmn/WaveFunctionCollapse fi. (30 Sep 2016). Retrieved May 20, 2017 from https://twitter.com/ExUtumno/status/781834584136814593

[9] Maxim Gumin. 2016. ConvChain. https://github.com/mxgmn/ConvChain, *GitHub repository* (2016).

[10] Maxim Gumin. 2016. SynTex. https://github.com/mxgmn/SynTex, *GitHub repository* (2016).

[11] Maxim Gumin. 2016. WaveFunctionCollapse. https://github.com/mxgmn/WaveFunctionCollapse, *GitHub repository* (2016).

[12] Maxim Gumin. 2017. WaveFunctionCollapse Readme.md. (18 May 2017). Retrieved May 20, 2017 from https://github.com/mxgmn/WaveFunctionCollapse/blob/master/README.md

[13] Taylor Holmes. 2016. Interview with phenomenal game designer Oskar Stålberg. (22 Jan 2016). Retrieved May 20, 2017 from https://taylorholmes.com/2016/01/22/interview-with-phenomenal-game-designer-oskar-stalberg/

[14] Ian D Horswill and Leif Foged. 2012. Fast Procedural Level Population with Playability Constraints. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*.

[15] Roland Kaminski, Torsten Schaub, and Philipp Wanko. 2017. A Tutorial on Hybrid Answer Set Solving with clingo. (2017). Retrieved May 20, 2017 from https://www.cs.uni-potsdam.de/~torsten/hybris.pdf

[16] Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. 2015. Understanding VSIDS branching heuristics in conflict-driven clause-learning SAT solvers. In *Haifa Verification Conference*. Springer, 225–241.

[17] Lin Liang, Ce Liu, Ying-Qing Xu, Baining Guo, and Heung-Yeung Shum. 2001. Real-time texture synthesis by patch-based sampling. *ACM Transactions on Graphics (ToG)* 20, 3 (2001), 127–150.

[18] Paul C Merrell. 2009. *Model synthesis.* Ph.D. Dissertation. University of North Carolina at Chapel Hill.

[19] Martin O'Leary. 2017. Twitter Bio. (2017). Retrieved May 20, 2017 from https://twitter.com/mewo2

[20] Joseph Parker, Ryan Jones, and Oscar Morante. 2016. Proc Skater 2016. (2016).

[21] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. 2016. *Choco Documentation.* TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. http://www.choco-solver.org

[22] Stuart J Russell and Peter Norvig. 2009. *Artificial Intelligence: A Modern Approach.* Prentice Hall.

[23] Adam M. Smith and Michael Mateas. 2011. Answer Set Programming for Procedural Content Generation: A Design Space Approach. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (Sept 2011), 187–200. DOI: http://dx.doi.org/10.1109/TCIAIG.2011.2158545

[24] Gillian Smith, Jim Whitehead, and Michael Mateas. 2011. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 201–215.

[25] Oskar Stlberg. 2017. wave.html. (18 May 2017). Retrieved May 20, 2017 from http://oskarstalberg.com/game/wave/wave.html

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

[27] Michael Walker. 2016. King James Programming. (2016). Retrieved May 22, 2017 from http://kingjamesprogramming.tumblr.com/

[28] Ryan Williams, Carla Gomes, and Bart Selman. 2003. On the connections between backdoors, restarts, and heavy-tailedness in combinatorial search. *Structure* 23 (2003), 4.