Automatic Generation of Game Content using a Graph-based Wave Function Collapse Algorithm

Hwanhee Kim
NCsoft
Seoul, Korea
greentec@ncsoft.com

Seongtaek Lee
NCsoft
Seoul, Korea
caracara@ncsoft.com

Hyundong Lee
NCsoft
Seoul, Korea
mansoul@ncsoft.com

Teasung Hahn
NCsoft
Seoul, Korea
spinel@ncsoft.com

Shinjin Kang
Hongik University
Sejong, Korea
directx@hongik.ac.kr

***Abstract*—This paper describes graph-based Wave Function Collapse algorithm for procedural content generation. The goal of this system is to enable a game designer to procedurally create key content elements in the game level through simple association rule input. To do this, we propose a graph-based data structure that can be easily integrated with a navigation mesh data structure in a three-dimensional world. With our system, if the user inputs the minimum association rule, it is possible to effectively perform procedural content generation in the three-dimensional world. The experimental results show that the Wave Function Collapse algorithm, which is a texture synthesis algorithm, can be extended to a non-grid shape with high controllability and scalability.**

***Index Terms*—Wave Function Collapse (WFC), Procedural Content Generation (PCG)**

I. Introduction

Procedural content generation (PCG) is a general term for systems that take in certain design patterns and output new assets from these patterns. PCG approaches include evolutionary searches, rule-based systems, and the instantiation of content from probability tables [1] [2]. Dahlskog et al. [3] proposed an evolutionary computation method for Super Mario Bros. using vertical slices of levels for the evolutionary computation. Alternatively, Snodgrass et al. [4] described a hierarchical method for procedurally generating maps using Markov chains. Their system takes as input a collection of human-authored two-dimensional (2D) maps and splits them into high-level tiles that capture large structures. Markov chains are then learned from those maps to capture and generate the structure of the level. Jain et al. [5] used autoencoders for game content generation, recognition, and repair, and described proof-of-concept implementations of autoencoders for these tasks for Super Mario Bros. levels. They train autoencoders to reproduce levels from the original Super Mario Bros. game and then use these networks to discriminate the generated levels from the original levels and to generate new levels via transformations from noise. Summerville et al. [6] presented a machine-learning technique to train generators on Super Mario Bros. videos, generating levels based on latent play styles learned from the videos. They compared the generated levels to the original levels and levels from a generator trained using simulated players. Shin et al. [7] created a system that uses PixelRNN to create a game level with play intent and a level editing tool that allows the user to edit the results interactively.

In computer graphics, texture synthesis is related to the problem of generating a large output image with a texture resembling that of a smaller input image. In many texture synthesis approaches, the input and output images are characterized in terms of the local patterns they contain, with these patterns typically consisting of sub-images only a few pixels in width. This approach has the advantage of offering better control because it can generate new and complex patterns with pattern sets that satisfy the small adjacency requirement.

Where Wave Function Collapse (WFC) departs from texture synthesis is a key aspect to enabling surprising new applications in game design. Karth et al. [8] demonstrated how WFC is a universally usable constraint solver and introduced the re-implementation of 2D bitmap WFC with answer set programming. In addition, they defined the adjacency relation in the existing 2D bitmap WFC as a positive adjacency relation and introduced the prohibited adjacency relation [9]. The WFC intuitively has an advantage in that the user can instinctively reflect the intention of the game designer in the PCG part by designing the tile and defining the linkage rules. Various indie games and developers have created game levels using WFC algorithms.[1] These applications demonstrate WFCs excellent content control ability.

In this paper, we propose a graph-based WFC algorithm by extending existing grid-based WFC rules, which were limited to 2D grid-based game content creation, to the graph world. Our results show that the proposed WFC algorithm can be applied to the Constraint Satisfaction Problem (CSP) and 3D worlds efficiently while preserving the association rule of game design.

II. Graph-based WFC Algorithm

The WFC is an algorithm released by the indie game developer Maxim Gumin through the Github Repository.[2] WFC divides an input pixel-based image or tile-based level

This work was supported by a National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2019R1A2C1002525).

978-1-7281-1884-0/19/$31.00 ©2019 IEEE

[1]For example, Caves of Qud, http://www.cavesofqud.com/, https://marian42.itch.io/wfc, and https://twitter.com/greentecq/status/1025348928634408960.

[2]https://github.com/mxgmn/WaveFunctionCollapse

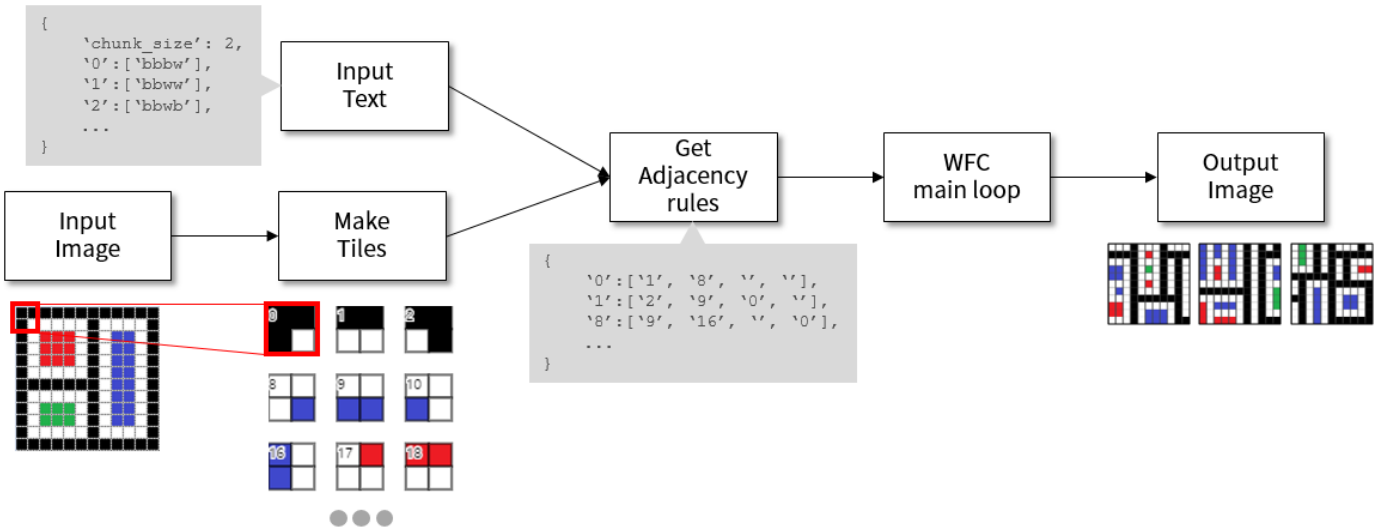

Fig. 1. Overview of the WFC algorithm process

into small chunks (1 1, 2 2 or larger) and rearranges them to create new levels. The newly generated level consists of overlapped or non-overlapped chunks. Figure 1 shows an overview of the WFC algorithm process. In this paper, we aim to extend the grid-based WFC algorithm to a graph-based type for application to unstructured data, such as a navigation mesh. The state space of the previous WFC is a regular grid, and each node constituting the grid has an equal number of neighbors. In contrast, the state space of a graph-based WFC has an unlimited and variable number of neighbors. Hence, we modified the propagation, compatibility, and backtracking processes in the original WFC algorithm. Figure 2 shows the main loop of the original WFC and graph-based WFC. The gray box highlights the difference between the two. For comprehensive details of the WFC algorithm, we refer the reader to [8].

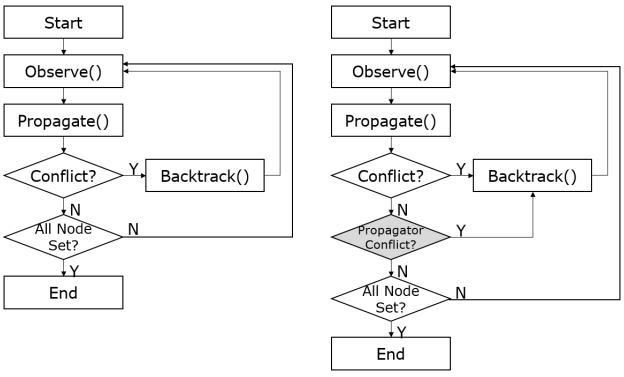

Fig. 2. Main loop of the original WFC (left) and graph-based WFC (right)

A. Handling input data

Both WFC and graph-based WFC receive input data and produce output. The input can be in the form of an image or text. It is assumed that the chunks attached to each other in the image are connected. The non-redundant set of chunks is called the tile, and it becomes the default unit of placement for the WFC. When the input is in text form, we need to define tile information in the same format as a JSON file. Graph-based WFC uses only text-type input. This is because a typical graph structure has a link that cannot specify a direction, unlike an image.

B. Graph and grid structure

A grid is a graph in which the number of neighbors of all nodes are the same. In other words, a graph is superset of a grid. In the regular rectangular grid, Sudoku game grid, and Voronoi non-grid[3] shown in Figure 3, the neighbors for an arbitrary cell are visualized. The number of neighbors of a regular rectangular grid is constant at four. The number of the Sudoku game grid's neighbors is fixed to 20, which includes 3 3 small grids within each cell, horizontal line, and vertical line. In contrast, the number of Voronoi non-grid neighbors is unlimited and variable. All of the above structures are applicable to graph-based WFC.

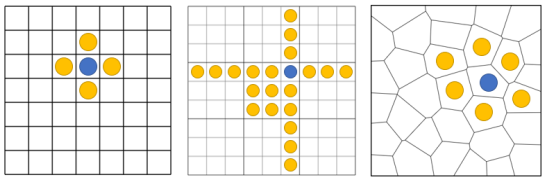

Fig. 3. Neighbors for arbitrary cells from a regular rectangular grid, Sudoku game grid, and Voronoi non-grid. The neighbors of the Sudoku game grid are determined by the game rules, and the neighbors of the regular rectangular grid and the Voronoi non-grid are determined by the adjacency of the edges. Yellow = neighbor cell.

C. Adjacency rule assignment

Figure 4 shows the graph-based adjacency rules used here and the corresponding WFC application results. The target

[3]A non-grid is the opposite of a grid. The number of neighbors of all nodes does not need to be equal.

map uses an unstructured lattice based on a Voronoi diagram instead of existing grid-type tiles. As described above, the algorithm is executed with the neighbor relation rule referring only to the connection relationships instead of the direction. As shown in Figure 4, node A can be connected to nodes A, B, and C; nodes B and C can only be connected to node A. The output of the graph-based WFC can be obtained such that nodes B and C are arranged in an area surrounded by node A.

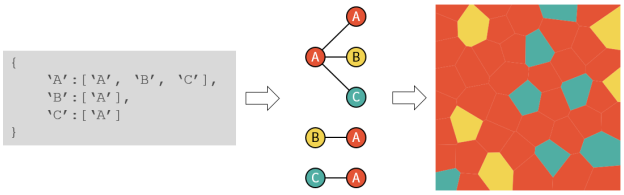

Fig. 4. Adjacency rules of the graph-based WFC algorithm according to arbitrary connection relationships and the corresponding output results

D. Modification of the Propagator and Compatible variables

To implement graph-based WFC, we modified the Propagator and Compatible variables of the original WFC. The Propagator variable is a 3D array in the original WFC. Let T be number of all tiles, D be number of possible directions, and V be number of connectable tiles (this is a variable number). Then, the size of Propagator P is the product of the three terms: $P = TDV$. In contrast, in graph-based WFC, we cannot take into account the direction, so P can be described as $P = TV$. Likewise, the Compatible C variable refers to the number of tiles that can be tied to a certain tile at a specific grid position, expressed as a 3D array of the number of all lattices N, T, and D in the existing WFC. Hence, the size of Compatible is the product of the three terms: $C = NTD$. It hence becomes a 2D array of N products. In graph-based WFC, it becomes T, because direction is not specified. Therefore, we can calculate the computation cost of C as $C = NT$.

E. Backtracking

In the WFC algorithm, the structure for iterating Observe() and Propagate() is the same until the full output is obtained. In addition, there is a Backtrack() function to deal with conflict when the value of Compatible is equal or less than zero on any cell—this means there is no neighboring cell that can be attached. However, in graph-based WFC, there is no direction check, so a connection that does not exist in the propagator can occur on the output, unlike in original WFC. We call this the "negative" case. Because this negative case does not fit the game designer's intent, we need to deal with this conflict. We can check whether the neighborhood of each cell whose value is set is within the range of the Propagator variable. If not, there is a "Propagator Conflict" as Figure 2 shows. In this case, we call the Backtrack() function. This function goes back to the point before the conflict occurs and runs the search again.

III. Experiment

A. Sudoku experiment

Sudoku is a number puzzle that must be filled with numbers from 1 to 9 in a 9 9 blank space. The criteria for filling in the numbers are that only one of the numbers 1 through 9 may be filled in for the vertical, horizontal, and small 3 3 grids without overlapping. There are many ways to solve Sudoku, and CSP is one popular method.[4] Figure 3 shows that each Sudoku cell has 20 neighbors, so this is a grid. However, we do not have to specify direction for these 1 to 9 tiles, so a graph-based WFC can be used to solve this problem.[5] Figure 5 shows how a Sudoku puzzle is solved on graph-based WFC. The leftmost column is a predefined value constraint, and graph-based WFC can solve the problem even with constraints. The unresolved values in the middle cells show the average color value of their possible outputs.

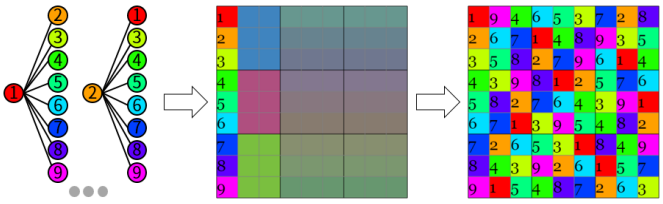

Fig. 5. Sudoku experiment. (Middle) Initial state (with manual constraints). (Right) Output.

B. Four-color theorem experiment

To verify the implemented non-grid WFC, we checked whether it could solve the four-color problem, which is used as a basic example in the CSP. The four-color problem involves dividing the plane into finite parts, then painting the parts that are in contact with each other with a different color. Four colors are sufficient to do this. In this problem, when a plane is divided into finite parts, each part has a non-grid structure in which the number of neighbors in each case is not constant. Therefore, this problem is suitable for verifying the basic performance of non-grid WFC in this paper. We want to ensure that we can set up four adjacency conditions and create a batch satisfying them. Figure 6 shows the results of the creation. In this experiment, we show that the system proposed in this paper does not need any additional constraints to solve the four-color problem and can solve the problem definition with the minimum definition, that is, with only the connections of each color tile.

C. PCG experiment with navigation mesh

Experiments were conducted to verify whether the graph-based WFC works properly on a 3D prototype game level. The navigation mesh was created using Blender. We applied an open-source pathfinding algorithm[6] to the level for extracting

[4]For example, see http://norvig.com/sudoku.html and https://gist.github.com/ksurya/3940679.

[5]https://twitter.com/greentecq/status/1037193248756957184

[6]https://github.com/donmccurdy/three-pathfinding