Puzzle-Level Generation with Simple-tiled and Graph-based Wave Function Collapse Algorithms

Hwanhee Kim, Beomjoo Seo, Shinjin Kang
School of Games, Hongik Univeristy
2639 Sejong-ro, Jochiwon, Sejong, Korea 30036
E-mail: greentec91@gmail.com, {bseo, directx}@hongik.ac.kr

***Abstract*—This study presents case studies using two wave function collapse (WFC) methods, graph-based WFC and simple tiled WFC, to create playable levels for two logic puzzle games: Strimko (Latin Squares) and Flow (connecting dots with pipes). We then evaluate the quality of the generated levels through extensive experiments. Our results indicate that WFC-generated levels are high quality, follow the graph structures' constraints, and are generated faster than levels generated by depth-first search and genetic algorithms. WFC methods can also adapt to new system specifications, common in puzzle games, by changing only the data instead of the code. This increases the stability of content production based on procedural content generation since it relies on data rather than procedures. Furthermore, WFC methods increase the efficiency of the manual process of creating in-game puzzle levels, allowing game designers to complete more tasks in the same amount of time and create a wider variety of assets.**

***Index Terms*—Procedural content generation, Puzzle game design, Wave function collapse algorithm**

I. Introduction

PUZZLE games are a popular genre that appeals to people of different ages and genders. Unlike action games that rely on quick reflexes and split-second decisions, puzzle games involve short or long periods of thinking and reasoning. Many puzzle games have been released and popularly enjoyed, from classic ones like Minesweeper and Solitaire to modern ones like the Candy Crush Saga series.

Puzzle game services need to provide players with levels that match their skill level. Game designers usually use tools related to paper planning and game engines to create a level. They also play many possible puzzle games at that level to see if they fit their plan. This can be very expensive in terms of development.

Researchers have proposed various procedural content generation (PCG) techniques to reduce development costs and create more diverse puzzle game levels [1]. These algorithms typically find optimal combinations that meet certain criteria, such as puzzles solvable by players of a specific skill level, with a specific length, or with a specific number of moves. Examples of PCG algorithms include Monte Carlo tree searches [2], Markov chains [3], answer-set programming solvers [4], and genetic algorithms [5].

Many of these techniques have been proposed by researchers, but they have seldom been used in real game development. This is because actual game developers must have a lot of prior knowledge to use these algorithms. Moreover, the algorithms often need to be changed if the game system changes because they depend heavily on the game system.

We propose using two wave function collapse (WFC) algorithms, graph-based WFC and simple tiled WFC, to address the problem of creating diverse and challenging puzzle levels. Using them, our intuitive interface in the game engine allows game developers to easily create puzzle levels that meet specific conditions, such as tile adjacency conditions and whitelisted/blacklisted tiles at specific positions, without requiring professional knowledge of PCG algorithms. This approach is also more adaptable to game system changes, requiring only small data changes and relatively fewer settings than traditional PCG algorithms. In this respect, our proposed method differs from PCG via Machine Learning (PCGML), a recently popular method of generating game content using machine learning [6], typically requiring tens to hundreds of existing data points for content creation.

II. Related Work

A. PCG-based puzzle-level generation algorithm

There are largely three approaches to generating puzzle levels, as illustrated in Figure 1 [7]. The first approach, *reverse search*, creates a puzzle level that meets the puzzle requirements. It then works backwards to find a start state that can be reached by applying playing actions in reverse order [8]. This method has the advantage of guaranteeing solvability of the generated level and being easy to implement. However, it has some limitations: it can only apply to puzzle games that have reversible actions and do not require complex design elements.

The second approach, *generate-and-test*, involves creating level components from scratch and then discarding those that do not meet the puzzle requirements [2]. To do this, a solver is used to check if the puzzle level can be solved, and if not, it is rejected. This method is suitable for puzzle games with complex requirements but does not guarantee the solvability of the generated levels. This method includes many PCG algorithms. PCGML can also be classified as generate-and-test since it requires checking solvability after generating.

Conversely, the third and final method, *constructive*, utilizes Markov chains [3] or answer-set programming solvers [4] to generate a puzzle level at once under given constraints. This method does not employ reverse engineering or iterative generation and testing of levels. Instead, it produces a complete, unbroken level in a single step. However, akin to the

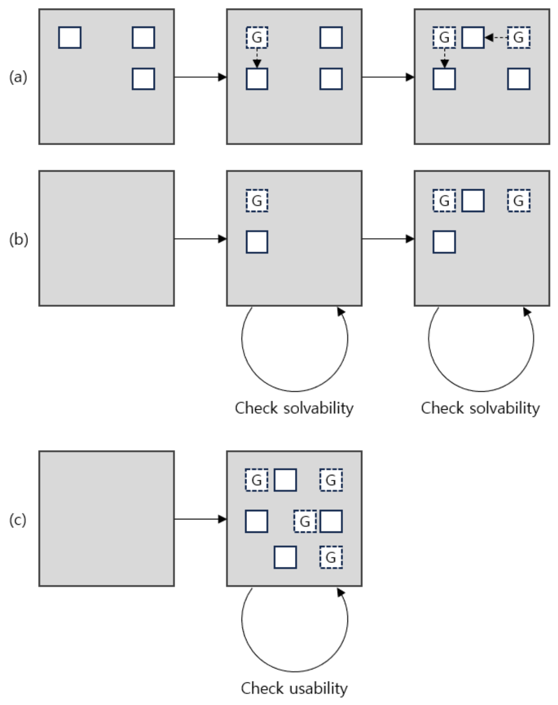


Fig. 1: Comparison of three puzzle-level generation methods. (a) reverse search: finding the start state after generating a solution in advance. (b) generate-and-test: placing level components starting from empty space; then, a solver examines whether the generated level can be solvable. (c) constructive: generating a level at once under given constraints, ensuring solvability; however, its usability requires further evaluation.

generate-and-test method, it necessitates an evaluation of the generated level's usability. While carefully defined constraints can yield playable levels, ensuring their quality and suitability often proves more challenging than with generate-and-test approaches.

Various generate-and-test techniques have been proposed in the game puzzle generation PCG field. Murase et al. applied a breadth-first search (BFS) test to a combination of random templates to generate levels for Sokoban [9]. Ashlock showed how an evolutionary algorithm can generate puzzle levels for maze games with simple and reversible movement rules [10]. Mantere et al. demonstrated how a genetic algorithm can generate challenging Sudoku puzzles [11]. Shaker et al. applied an evolutionary computation technique to a physics-based puzzle game [12]. Fatemi et al. designed Sudoku levels with varying difficulty by solving a constraint-satisfaction problem that guarantees a unique solution [13]. Rychnovsky made a PCG algorithm that made different levels for Fruit Dating, a commercial puzzle game. The algorithm puts fruits, obstacles, and walls on a 2D grid. He tested the quality of the levels by applying a BFS test to random batches [14]. As previously noted, generate-and-test methods rely on a combination of random generation and BFS test or an evolutionary computation algorithm to create puzzle levels. However, these methods have primarily focused on Sudoku and maze games.

These PCG-based puzzle level generation methods apply to classic puzzle games. However, recent puzzle games often require complex or novel user interfaces (e.g., drawing). These games need a different methodology that can handle higher-dimensional spatial data and newer interfaces. In this study, we show that WFC, a PCG algorithm that uses graphs to represent constraints, can be applied in the generate-and-test field; it outperforms the existing random arrangement and evolution algorithm.

PCG algorithms that generate levels by modelling or learning adjacencies from existing data, such as GANs [15] and autoencoders [16], are also a type of generate-and-test. However, these methods require level data with tens to hundreds of guaranteed gameplays. Conversely, the method presented in this article requires very little existing level data.

Our proposed technique works with both traditional logic games like Strimko and modern nonlinear games like Flow that require a drawing interface. Unlike existing tile-based methods, our technique can handle higher-dimensional spaces and generate levels faster than the existing depth-first search (DFS) and genetic algorithm (GA) verification methods. We also examine the diversity of our generated levels using the clustering analysis.

B. *WFC-based commercial game-level generation*

The WFC algorithm was disclosed by Maxim Gumin, an independent game developer, who shared it on GitHub [17]. To generate bitmap images, it uses a similar method as TextureSynthesis [18] and ConvChain [19], two PCG algorithms that Gumin introduced for generating textures. While the two PCG algorithms use the entire bitmap as input, WFC relies only on the connection relationship between chunks of pixels. This makes WFC more suitable for creating sparse data when only a few data points exist in a large state space. Therefore, WFC can produce a combination of pixels with correlations even with a few image inputs. Moreover, it can create a combination of pixels that satisfy constrained rules, which can be used for generating puzzle levels that meet preconstrained conditions.

Maxim Gumin proposed two WFC methods: the simple tiled and overlapping models. The simple tiled model defines tiles and their connection rules using a data file, generating an image based on this information. Conversely, the overlapping model segments a given image into overlapping tiles, establishing connections between these segmented tiles and producing a similar image. WFC can generate 2D tiles for role-paying game (RPG) dungeons [20] and 3D meshes for strategy game maps [21]. Deepmind used 2D WFC as a height map to create a 3D mesh stadium for reinforcement learning [22]. These examples use WFC on a 2D or 3D grid structure. However, both methods are restricted to a grid with a fixed number of neighbors.

To address this limitation, the graph-based model [23] [24], an extension of the simple tiled model, defines a set of predefined tiles and their connections in a data file. This model operated on a graph structure, constructed based on the logical connections between the tiles, regardless of their physical adjacency, and compared it with other graph coloring algorithms [25]. These logical connections enable puzzles to incorporate a variable number of logically neighboring cells, which will be detailed later in this article. Since generating puzzle levels with different constraints and requirements, such as Strimko and Flow, is relatively uncommon, we will explore the quality and generation speed of generating these puzzle levels using the WFC algorithms. The simple-tiled and graph-based models were used to create the puzzle levels in these two games, both of which are characterized by producing content based on rules.

Constraint satisfaction problems (CSPs) are a broad class of problems that involve finding complete states that satisfy a set of constraints. CSPs can be used to model various real-world problems, such as scheduling, resource allocation, and configuration. Many Constraint solver algorithms commonly include constraint propagation, backtracking, and local research. WFC is a recent technique for CSPs that specializes in procedural content generation, such as game-level generation.

It is closely related to model synthesis, proposed by Paul C. Merrell [26]. Model synthesis uses arc consistency (AC)-3 [27] or -4 [28] algorithms for constraint propagation, a method that inspired Maxim Gumin to develop WFC. WFC and model synthesis are similar in that they extract patterns from input and relationships from data or specifications. However, WFC differs by (1) being more specialized for generating content with complex or irregular structures, (2) focusing on efficiency (i.e., generation speed), and (3) having some unique features, such as the “lowest entropy heuristic,” which removes directional bias in generated results and is well-suited for pre-constrained problems. WFC’s adaptation of the AC-3 algorithm makes it a valuable tool for pattern generation that differs from the standard AC-3 approach and highlights the flexibility and adaptability of WFC for a broader range of applications in the field of procedural content generation.

III. Method

The original WFC has two model implementations: simple tiled and overlapping. The simple tiled model divides the input image or voxel model into non-overlapping tiles or chunks and stores their connection rules. It can also take text inputs for connection rules and use the WFC algorithm to place tiles. The simple tiled model is named so because it stores the appearance probability of a tile rather than a rule. In contrast, the overlapping model divides the input image or voxel model into overlapping tiles or chunks and stores their connection rules and appearance probabilities.

Most existing WFC implementations employ a grid structure composed of cells, which are uniformly sized, empty shapes that can be congruent and parallel to each other and possess the same number of neighbors. Leveraging the WFC algorithm, each cell is systematically populated with one of the target tiles during content generation.

Unlike these WFCs, which are implemented on a grid where each node has the same number of neighbors, the graph-based WFC operates on a graph. The graph-based WFC allows each cell to have a variable number of neighbors, unlike the grid-based WFC, which assumes a fixed number of neighbors for each node. The grid-based WFC stores the relative position and direction (top, bottom, left, right) of each tile and their connection rules. The graph-based WFC does not need directions because it works on simple undirected graphs. Figure 2 illustrates the types and connections of tiles used in the simple tiled model and how they are placed. Graph-based WFC is based on the simple tiled model.

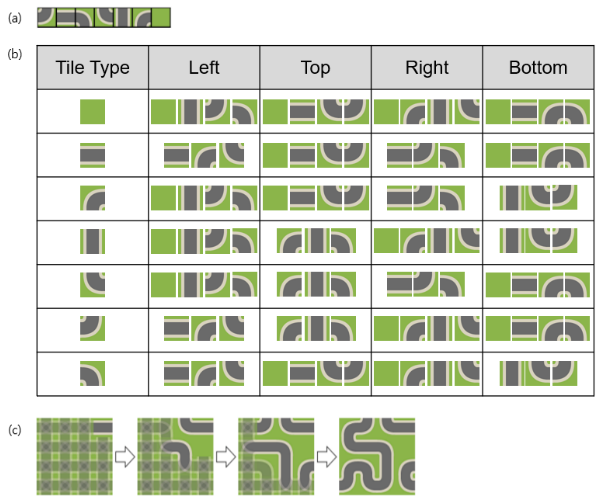


Fig. 2: WFC simple tiled model example. (a) Seven types of tiles, (b) The connectivity of the tiles. Each row for individual tile types shows which tiles can be connected to the left/above/right/bottom of each tile, (c) Resulting image that gradually becomes clearer when tiles are placed.

This study explores how to generate puzzle levels using WFC. WFC requires strict constraints to create puzzle levels that match the intended design. The constraints include the type of tiles that can be used and the rules that can be applied to each cell. The WFC algorithm takes an input with a predefined connection relationship and observes an uncertain cell to determine its state. Then, it propagates the tiles compatible with the observed tile to the neighboring cells and repeats the observation process until every cell has a single tile.

Conversely, the overlapping WFC model necessitates content generation based on existing images or data, which is not well-suited for puzzle-level production because of the unlikely coverage of all puzzle rules by a single image or data point. While multiple images or data could be used with the overlapping model, this approach is not pursued in this study owing to the resulting complexity of rules and reduced computational efficiency compared to the simple-tiled and graph-based models.

For both the overlapping and simple tiled models, the navigation of tiles follows a consistent process. Figure 2(c) offers a visual glimpse into this process, where blurred tiles within a cell signify multiple placement possibilities, while a single clear tile denotes a fixed choice. Now let’s delve into a more detailed explanation of how the WFC algorithm operates.

Initially, the algorithm commences by defining a set of permissible tiles and creating an empty grid, where each position holds a wave function encompassing all potential tiles. Subsequently, it identifies cells with the least uncertainty, considering entropy, and evaluates tiles within the wave function compliance with constraints established by neighboring tiles that have already undergone collapse. Incompatible tiles are systematically removed from the wave function. If only one tile remains in the wave function after this evaluation, the algorithm collapses the position to that tile, finalizing the output decision for that specific position.

Following each collapse, the algorithm updates the wave functions of neighboring positions to incorporate the recently collapsed tile, adjusting possibilities based on the imposed constraints. This iterative process repeats the observation and the propagation steps until all positions within the grid have undergone collapse, making the completion of the generation process.

Sometimes, placing tiles can cause contradictions with connection rules and make it impossible to place more tiles. The original WFC algorithm restarts the whole process when this happens. However, this can be very costly if there are many connection rules. Therefore, an additional feature called backtracking has been implemented in the WFC algorithm and the graph-based WFC. Although backtracking was not an integral part of the original WFC algorithm, it was introduced as “modifying in blocks” in model synthesis, thus contributing to its faster and failure-free performance compared to WFC in all runs during algorithm comparisons [29]. Thus, we employ backtracking when placing tiles becomes impossible due to discrepancies between tile connectivity and their current placement. All previously selected final tiles are saved to implement backtracking but only when each cell contains a single tile. Subsequently, the state of all cells is reset, and the saved final tile information is reintroduced into the cell where it was initially selected. This effectively reverts the process to a state preceding the contradiction, allowing for the selection of alternative tiles.

The appearance probability of tiles is a crucial factor that influences the output. This study shows how changing the appearance probability affects the generation of Flow puzzle levels.

Another factor that affects the output is the constraint on the tile type that can be placed in each cell. The constraint can either allow or forbid a specific list of tiles to be placed at a specific location. This can be simplified as allowing only a specific list of tiles to be placed at a specific location because the types of tiles are finite. Constraints are set before running the algorithm and are propagated to all cells. This study demonstrates how to use constraints to generate Flow puzzle levels.

IV. Experiment

A. Creating a Strimko puzzle

Strimko is a puzzle game by the Grabarchuk Family, created in 2008. It is based on Latin Square, a concept by 18th-century mathematician Euler. It is similar to Sudoku. The goal of Strimko is to fill an N × N grid with numbers from 1 and N. Each number must appear only once in each column, row, and stream. A stream is a group of N connected numbers without repeats. There are N streams in an N × N grid. Figure 3 illustrates three streams in a 3x3 Strimko grid. In Strimko gameplay, parts of the puzzle are hidden, and the player must use the visible numbers to deduce the hidden ones.

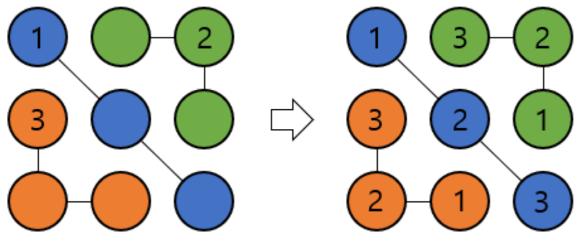


Fig. 3: Three streams of a 3 × 3 Strimko puzzle distinguished by color.

The original web-based game, which is no longer available, had puzzles with grids of 4 × 4, 5 × 5, 6 × 6, and 7 × 7 grids [30]. The game could also have grids larger than 8 × 8. A book called Chain Sudoku has the same rules as Strimko. It has 200 puzzles with an 8 × 8 grid [31].

This study tests puzzle level generation for three grids: 4 × 4, 5 × 5, and 6 × 6. We used a NodeJS program (version 14.15.4) on a Windows 10 PC with an i7-9700 CPU, RTX 2080Ti, and 64GB RAM for all experiments.

We need two steps to generate a puzzle: 1) make a valid Strimko grid and 2) make valid Strimko puzzles. A valid Strimko grid has N streams on an N × N grid. Each stream has N cells with no repeats. The streams can connect in eight directions: up, down, left, right, and diagonal. A simple way to make a valid Strimko grid is to use only horizontal or vertical streams. But for more complex puzzle grids, we need to search. We start from an empty random cell on the grid and move in eight directions. We find empty cells and add them to the stream. When the stream has N cells, we start a new stream from another empty random cell. If we cannot find any empty cells, we go back to the previous cell. If we finish the search normally, we sort the streams by the number of nodes they have. Figure 4 illustrates an example of backtracking during the search for a Strimko grid. Figure 5 shows the grids of various sizes generated by the search.

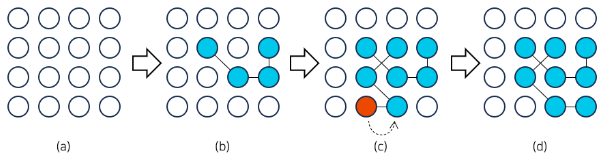


Fig. 4: Search example of the Strimko puzzle grid. (a) An empty 4 × 4 grid, (b) First stream found. Search complete. (c) A situation in which searching becomes impossible when the red circle occurs during the second stream, requiring backtracking. (d) Search in a different direction by backtracking, completing the second stream.

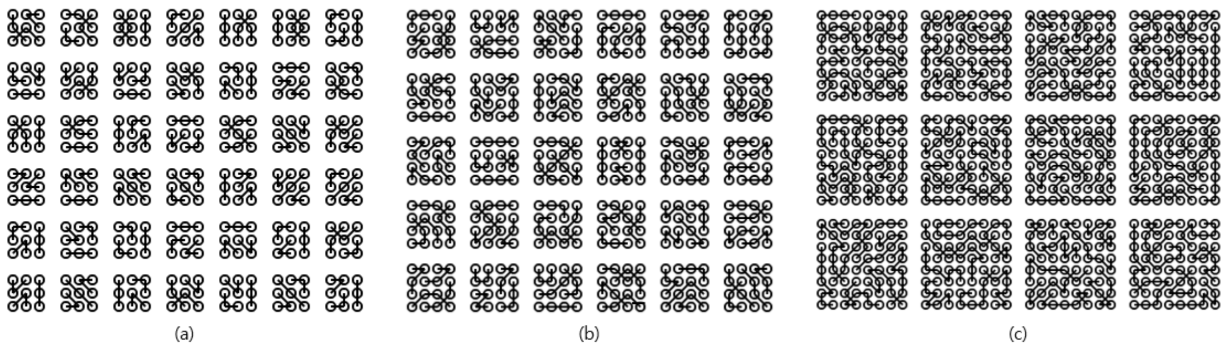


Fig. 5: Valid Strimko puzzle grid. (a) 3 × 3 (b) 4 × 4 (c) 8 × 8.

Then, we create a valid Strimko level using our graph-based WFC on a valid Strimko grid. The Strimko game rules require that the neighbors of a given cell, which are the cells that share the same column, row, or stream, have different numbers. Figure 6 shows an example of neighboring cells (in pink) for a cell (in blue) in different grids. Thus, the number of neighbors can vary from 7 to 8 (illustrated in Figure 6) depending on the grid. We use our graph-based WFC to specify the cell relationships since the original grid-based WFC method cannot capture this variation.

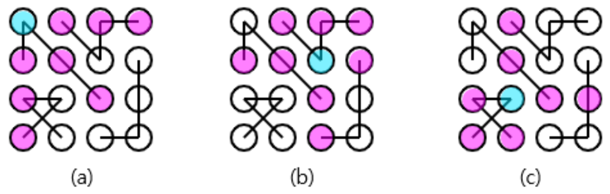


Fig. 6: Visualization of the neighbors (in pink) of each cell (in blue) in a Strimko puzzle level. (a) The number of neighboring cells of the upper left cell is 8. (b) The number of neighbor cells of the cell in column 3 row 2 is 8. (c) The number of neighbor cells of the cell in column 2 row 3 is 7.

The tiles in a Strimko puzzle are numbered from 1 to N. The connection relationships of the tiles are defined such that a number can meet all other numbers without meeting itself. The connection relationships are defined for a 4 × 4 Strimko puzzle level, as shown in Figure 7.

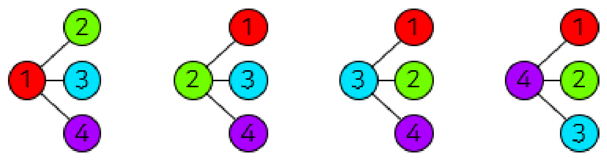


Fig. 7: Definition of connection relationships in a 4 × 4 Strimko puzzle level.

Graph-based WFC can only generate valid Strimko puzzle levels from some Strimko puzzle grids. Searching all possible cases is infeasible except for 3 × 3 grids because of computational limitations. Table I shows that the success rate of generating a valid Strimko puzzle level from a valid Strimko puzzle grid for a week-long experiment is about 62-74% for grids up to 7 × 7. However, this rate drops significantly to 47.4% and 23.8% for grids of size 8 × 8 or larger. Figure 8 illustrates examples of successful and failed generations for 3 × 3 and 8 × 8 grids. In the 3 × 3 grid example, not all numbers can be placed according to Strimko's rule. For example, if we place a 1 on the diagonal, we cannot place a 2 anywhere without creating a stream with two 2s, which is invalid.

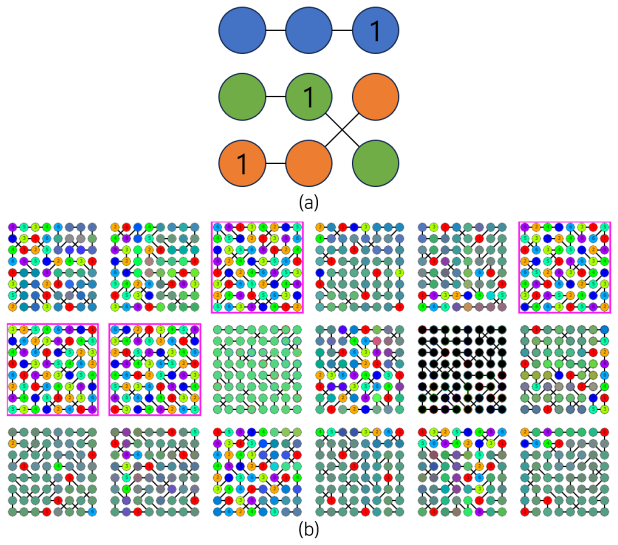


Fig. 8: Strimko puzzle level search results example. (a) 3 × 3 grid. (b) 8 × 8 grid. Only the four valid puzzle grids marked in pink are valid puzzle levels; searching failed for the others.

We evaluated the performance of our graph-based WFC method against depth-first search (DFS) and the genetic algorithm (GA) on identical puzzle grids, utilizing 4 × 4, 5 × 5, and 6 × 6 grids as target levels. These three algorithms exhibit distinct characteristics that make them suitable candidates for comparison in the context of solving the CSP problem of generating puzzle levels. DFS is a straightforward and memory-efficient approach but may encounter scalability challenges. GA excels in handling large solution spaces and complex constraints; it necessitates parameter tuning, does not guarantee an optimal solution, and can also exhibit slow performance as the search space expands. WFC may be slower

	3 × 3	4 × 4	5 × 5	6 × 6	7 × 7	8 × 8	9 × 9	10 × 10
Number of valid Strimko puzzle grids	132	67,214	1,541,516	648,888	408,340	79,279	6,497	3,534
Number of valid Strimko puzzle levels	92	42,118	978,950	439,476	305,150	37,600	1,547	88
Ratio of valid puzzle levels per valid puzzle grid	69.7%	62.7%	63.5%	67.7%	74.7%	47.4%	23.8%	2.5%

TABLE I: Success rates of valid Strimko puzzle levels for valid Strimko puzzle grids

than DFS and GA due to its requirement for pre-established knowledge. However, WFC can effectively manage large solution spaces once this knowledge is in place, significantly reducing the number of nodes to explore compared to the other two algorithms.

For DFS, we employed a straightforward approach of sequentially placing numbers on unoccupied nodes until the correct solution was identified. This process was conducted in a randomized manner to prevent similar numbers from overlapping in similar locations, leading to a marginal improvement in search efficiency.

In the case of the GA, we initialized the starting population with randomly placed numbers. A predefined percentage of the elite population with high fitness scores was retained, while the remaining individuals were regenerated by copying from the elite population. This process was iterated by repeatedly performing the mutation operation of swapping two numbers within a random row. Subsequently, we evaluated whether each row, column, and stream contained unique numbers, and the fitness score was decremented by the number of duplicate numbers (e.g., in a 5 × 5 board, 1,2,3,4,5 has fitness=0,1,2,3,4,4 has fitness=-1, and 1,2,2,2,2 has fitness=-3). We employed population sizes of 32, 64, and 128, respectively, to adapt to the increased board size. In addition, the exploration node counts were scaled up to 100,000, 200,000, and 400,000, respectively. To further enhance the algorithm's performance, we preserved 50% of the elites in the next generation, ensured that each row contained unique numbers, and incorporated number position swaps within the mutate function to minimize unnecessary exploration. Conversely, the DFS algorithm relied on node exploration without such parameter adjustments.

Table II lists the average times needed to generate 100 levels from a valid Strimko puzzle grid, repeating 10 times under the same conditions. Our graph-based WFC was faster than DFS and GA on a grid of size 5 × 5 and larger. It was also more stable, with similar execution times and a narrow confidence interval. For a 4 × 4 board, the problem space has 64 possible configurations (4 possible numbers for each of the 4 × 4 nodes), limiting the number of nodes to explore. Consequently, DFS, which randomly explores nodes without considering constraints, appears to be the fastest algorithm. However, for a 6 × 6 board, the number of nodes to be explored increases by 237.5% (i.e., $6^3 = 216$ configurations). While the exploration time of DFS and GA increases exponentially with the problem space size, WFC does not significantly increase the number of nodes to explore because it already possesses information about invalid node combinations in the form of connection relationships. This results in a sublinear increase in execution time for WFC compared to the exponential growth observed in DFS and GA.

Algorithms	4 × 4	5 × 5	6 × 6
Graph-based WFC	16.96±1.46	**41.74±3.63**	**51.80±3.63**
DFS	**7.81±0.45**	75.09±4.48	1376.04±97.03
Genetic Algorithm	26.52±5.31	163.53±11.16	1053.87±194.64

TABLE II: Comparison of creation times (in seconds) of 100 Strimko puzzle levels for each algorithm

Algorithms	4 × 4	5 × 5	6 × 6
Graph-based WFC	1,560	1,592	1,457
DFS	1,779	7,639	114,728
Genetic Algorithm	1,706	2,392	4,009

TABLE III: Comparison of the number of Strimko puzzle grids required by each algorithm to create 1000 Strimko puzzle levels

Another metric to consider is the number of Strimko puzzle grids used by each algorithm to generate 1,000 Strimko puzzle levels. We used the same pre-made Strimko puzzle grids for all algorithms, which affects the number of Strimko puzzle grids needed. Graph-based WFC does not change the number of grids needed even when the grid size increases, unlike DFS, which increases the number of grids dramatically, as shown in Table III.

B. Generating a Flow puzzle

Flow is a mobile puzzle game by Big Duck Games, released in June 2012. It is based on an older puzzle called Numberlink, which was first mentioned in a column in 1897 [32] and a mathematical problem in 1917 [33]. We focus on the Flow puzzle because it has almost the same rules as Numberlink. The Flow puzzle aims to connect pairs of nodes with different colors on an N × N grid. The connections must be unbroken lines that do not cross each other. Each cell should have only one line in it. The puzzle is solved when all nodes are connected and the grid is full. Figure 9 illustrates examples of a Flow puzzle and a Numberlink puzzle.

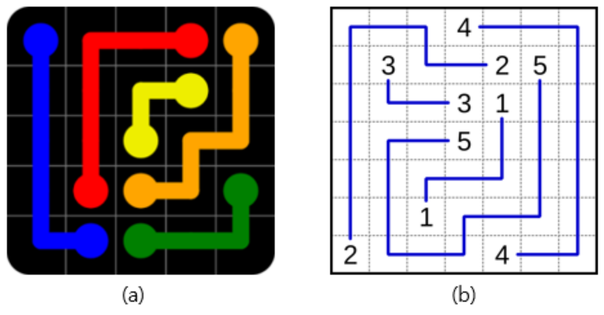


Fig. 9: Sample puzzles (a) Flow (b) Numberlink.

We initially considered using graph–based WFC to generate Flow puzzle levels. However, after careful scrutiny, we decided that the simple tiled model would be more suitable owing to the fact that each tile in Flow has only four neighbors: top, bottom, left, and right. This makes the simple tiled WFC possible for Flow puzzle level generation, unlike for Strimko puzzles.

The Flow puzzles have tiles of different colors, but we can only generate puzzle levels with one color tiles. This is because we separate the lines after generating the puzzle level and then paint them with different colors. Figure 10 illustrates how we create a flow puzzle with one color and assign a different color to each flow.

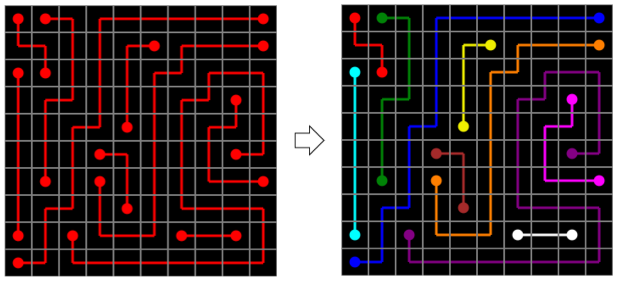


Fig. 10: Postprocessing after generating a Flow puzzle.

Figure 11 shows the 11 tile types in a Flow puzzle: four dead ends, four corners bent at 90 degrees, horizontal, vertical, and empty. A Flow puzzle space cannot have empty cells. So, we do not need empty tiles for the puzzle level. However, we define empty tiles as a tile-type constraint, which we explain below.

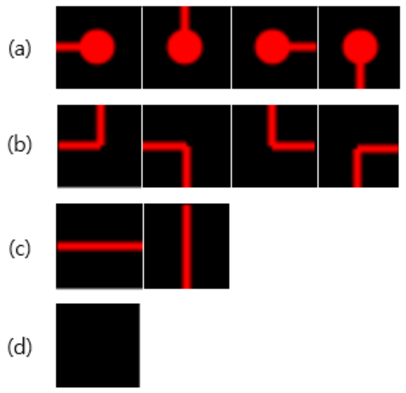


Fig. 11: Tile types of a Flow puzzle: (a) dead end, (b) corner, (c) straight (horizontal, vertical), and (d) empty.

We set the connection rules between these tiles so that lines or empty spaces match each other. However, we do not allow dead ends to connect directly because that would make the puzzle too easy. We also do not allow corner tiles to connect and bend at 180 degrees because that would break the constraint that all cells must be used. If we allow such a connection, there might be a shortest-distance solution that skips the corresponding tile and violates the puzzle constraint. Figure 12 illustrates an example of a valid connection and two invalid connections for the Flow puzzle.

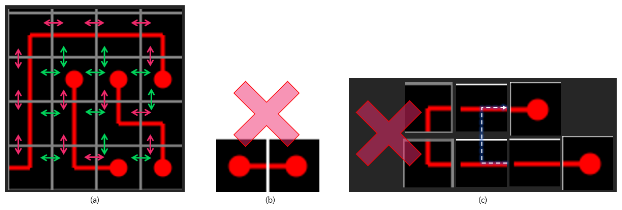


Fig. 12: Connection relationships between Flow puzzle tiles: (a) red arrows indicate a connection between lines, and green arrows indicate a connection between empty spaces without lines, (b) direct connection between dead ends is prohibited, (c) corner tiles that connect each other and bend at 180 degrees are prohibited.

We need two more constraints on the types of tiles that can go in each cell to get a normal Flow puzzle. The first constraint is to put only empty tiles outside the puzzle level. This way, only tiles that can connect to the empty tile can go into the edge cell of the puzzle level. This makes the line connection stay within the puzzle level and prevents any broken line from going out. We do not actually put empty tiles outside the puzzle level. We just add this connection constraint when we run the WFC algorithm. The second constraint is to avoid putting an empty tile in the puzzle-level space if it is already assigned as a tile type. This means that we only use 10 types of tiles that are not empty in the puzzle-level space. Figure 13 shows these two tile constraints.

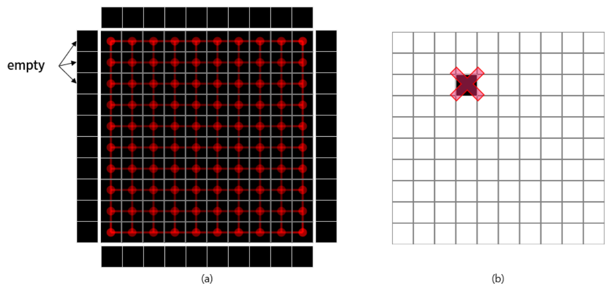


Fig. 13: Constraints of tile types for a Flow puzzle: (a) forcing empty tiles to be placed outside the puzzle level and (b) prohibiting empty tiles inside the puzzle level.

Unlike the Strimko puzzle, the Flow puzzle can adjust the appearance probability of each tile. In the Strimko puzzle, each tile must have a different number to constitute a valid puzzle level. Consequently, changing the appearance probability does not affect the number of tiles. In Flow, however, we can generate different levels by changing the appearance probability of tiles because Flow is more flexible. Figure 14 illustrates how adjusting the tile probability can change the outcome dramatically. Table IV lists the statistics of the features of a 10×10 Flow puzzle level generated with different appearance probability settings. Maps with 50 times as many straight tiles or 50 times as many corner tiles have a larger average length of flow and a smaller number of flows than maps where all tiles have the same probability of occurrence. In general, the fewer flows and the longer the length of a single flow, the

Features	equal for all tiles	50 × more straight tiles	50 × more corner tiles
Length of Longest Flow	10.20	40.43	18.25
Length of Shortest Flow	3.00	3.80	3.00
Difference of Longest, Shortest	7.20	36.63	15.24
Average Length of Flow	4.54	15.23	7.47
Standard Deviation of Flow Length	1.91	14.00	4.50
Count of Flow	22.17	5.66	13.63

TABLE IV: Average values of the features of 1,000 10 × 10 Flow levels generated by the simple tiled WFC algorithm

more difficult the level is perceived to be, so adjusting the probability of different tile types enables you to create more or less difficult maps.

In addition, the ability to adjust tile probabilities can be used to fine-tune the difficulty of puzzles in border game contexts or to create challenges that demand unique skills, such as designing levels with destructible walls that can only be breached through the use of bombs. In game prototyping, where there is often frequent experimentation and game designers may need to create, remove, and test a lot of content in a short amount of time, the WFC algorithm can be used to generate different maps for these experiments, which can then be tested and used to refine rule sets and constraints. This iterative process, coupled with the WFC algorithm's ability to quickly generate a multitude of game maps with various elements, makes it invaluable in game development, especially when striving for rapid, data-driven improvements and adjustments in the quest for a more engaging and finely-tuned game experience.

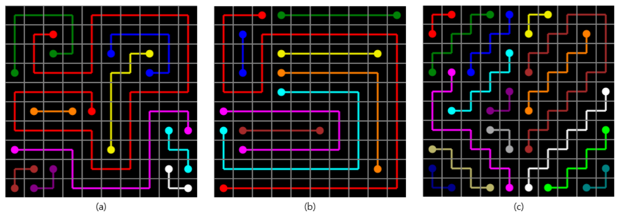


Fig. 14: Examples of tile appearance probability adjustments in a Flow puzzle: (a) equal probability for all tiles, (b) 50 times more straight tiles, and (c) 50 times more corner tiles.

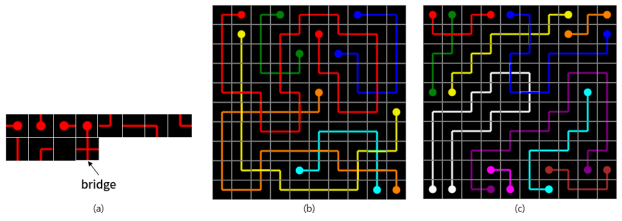


Fig. 15: Examples of Flow puzzles when adding a bridge tile: (a) bridge tile type, (b) equal appearance probability for the bridge tile and existing tiles, and (c) $\frac{1}{10}$ appearance probability for the bridge tile compared to existing tiles.

The WFC algorithm creates content based on data. So, it can handle specification changes and additions that often happen in puzzle games. For example, Big Duck Games, the maker of Flow Free, has made the game Flow Free: Bridges with a bridge that connects two different lines. WFC can do this by just adding a bridge tile. Furthermore, as shown in Figure 15, we can get different results by changing the appearance probability of a bridge tile.

Algorithms	4 × 4	5 × 5	6 × 6
Simple tiled WFC	**0.14±0.00**	**0.16±0.00**	**0.23±0.03**
DFS	1.70±0.17	4.38±0.90	13.68±2.82
Genetic Algorithm	60.85±5.41	274.35±20.74	4106.51±564.90

TABLE V: Comparison of creation times (in seconds) of 100 Flow puzzle levels for each algorithm

Features	Simple tiled WFC	DFS	GA
Length of Longest Flow	9.95	7.67	10.48
Length of Shortest Flow	3.21	3.03	3.25
Difference of Longest, Shortest	6.74	4.64	7.23
Average Length of Flow	5.34	4.41	5.60
Standard Deviation of Flow Length	2.39	1.62	2.60
Count of Flow	6.96	8.3	6.67

TABLE VI: Average value of 6 × 6 Flow level features used in t-SNE.

We generated the same Flow puzzle levels using WFC, DFS, and GA to compare the performance. The target levels were 4 × 4, 5 × 5, and 6 × 6 grids. For DFS, we employed a simple method of placing tiles one by one on unoccupied nodes and exploring until the correct solution was identified. This process terminates upon encountering an invalid placement in the Flow puzzle depicted in Figure 12.

The overall methodology for the GA closely resembles that used for the Strimko puzzle. A random tile is placed in the initial population, followed by the preservation of elite tiles and mutation to a random tile. The fitness score penalized tiles whenever they fail to connect to adjacent tiles or when an invalid placement occurs, as illustrated in Figure 12. The edges of the map were employed to restrict the permissible tiles, akin to the WFC algorithm, to expedite the search.

Table V lists the time needed to generate 100 non-overlapping levels from scratch. On all grid sizes, the simple tiled WFC algorithm was faster than the DFS algorithm and GA. The problem space size is $4 \times 4 \times 11 = 176$ for 4 × 4 (where 11 is the number of tiles) and $6 \times 6 \times 11 = 396$ for 6 × 6. Similar to Strimko, DFS, and GA exhibit an exponential increase in execution time as the problem space size grows, while WFC demonstrates a sub-linear increase in execution time. This is because the number of nodes to explore does not increase significantly for WFC since it possesses information in advance about invalid node combinations (forbidden tiles and allowed connections).

We used t-SNE to visualize the diversity of some features on the generated Flow puzzle levels. We calculated and used six features for clustering: the longest flow length, the shortest flow length, the difference between the longest and the shortest flow lengths, the average flow length, the standard deviation of flow length, and the number of flows. Table VI lists the average value of each feature. In terms of features, the DFS tends to generate shorter levels with more flows, while the GA tends to generate longer levels with fewer flows. The simple tiled WFC falls somewhere in between the two algorithms. In terms of play experience, longer-than-average flows make the game harder, so WFC and GA are more likely to create

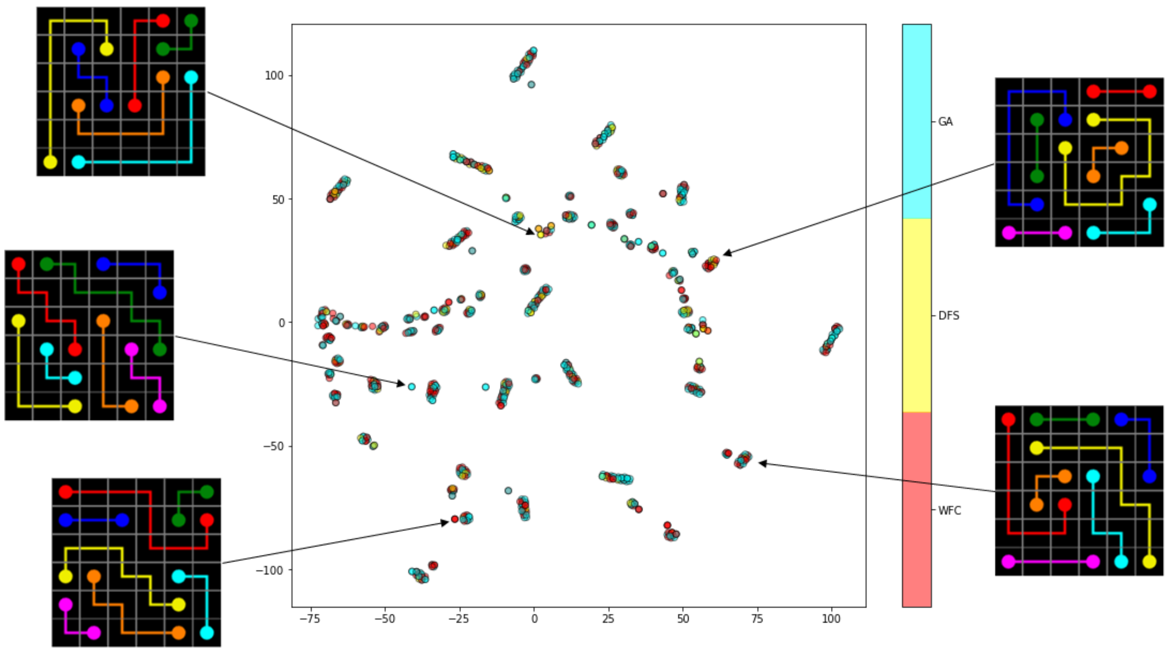


Fig. 16: A cluster of 6 × 6 Flow puzzle levels visualized with t-SNE.

difficult levels than DFS.

Figure 16 exhibits levels generated by three different algorithms: WFC (red), DFS (yellow), and GA (cyan). Each algorithm generated 10,000 levels, but we removed overlapping data points to avoid redundancy, resulting in 351 levels for WFC, 112 for DFS, and 425 for GA. In terms of clustering, we notice that DFS-generated levels are distributed across fewer clusters than WFC- anWd GA-generated levels. Examining the levels within each cluster reveals variation in the presence of unusually long flows and the total number of flows.

Figure 17 leverages Kernel Density Estimation (KDE) [34] to compare map distributions produced by the three algorithms, offering a detailed analysis beyond t-SNE's dimensional reduction. In the figure, DFS, highlighted in orange, displays a notably narrower distribution than the other two algorithms. When the length of shortest flow for DFS is 3, the longest flow length typically falls at a lower value, while the flow tends to be higher. This indicates that DFS produces maps with shorter lengths and more lines, aligning with our observations in Table VI. Remarkably, WFC and GA demonstrate similar distribution patterns. However, a crucial distinction emerges in their processing speed; GA significantly lags behind WFC when generating maps larger than 6 × 6. This performance advantage positions WFC favorably for tackling larger map generation tasks.

V. Conclusion

We used the graph-based and simple tiled WFC algorithms to generate levels for Strimko and Flow puzzles and evaluated the quality of the levels. These WFC algorithms outperformed DFS and GA in terms of level generation speed and generated more diverse Flow puzzle levels than DFS. Using WFC for puzzle level generation also makes it easier for content creators to use PCGs, as the core part is changed from procedure units to data. In game prototyping, the WFC algorithm's swift generation of diverse game maps facilitates rapid and iterative experimentation, which is invaluable for achieving data-driven improvements.

Acknowledgement

This research was supported by the Culture, Sports, and Tourism R&D Program through the Korea Creative Content Agency grant funded by the Ministry of Culture, Sports and Tourism in 2022 (Project Name: Development of artificial intelligence-based game simulation technology to support online game content production, Project Number: 202201590001).

References

[1] De Kegel, B.; and Haahr, M. 2019. "Procedural puzzle generation: A survey." *IEEE Transactions on Games*, 12(1): 21–40.
[2] Kartal, B.; Sohre, N.; and Guy, S. 2016. "Generating sokoban puzzle game levels with monte carlo tree search." *The IJCAI-16 Workshop on General Game Playing.*
[3] Snodgrass, S.; and Ontañón, S. 2016. "Controllable Procedural Content Generation via Constrained Multi-Dimensional Markov Chain Sampling." In *IJCAI*, 780–786.
[4] Lindeman, D. 2018. "Puzzle level generation with answer set programming." *Technical Library.*
[5] Amos, M.; and Coldridge, J. 2012. "A genetic algorithm for the Zen Puzzle Garden game." *Natural Computing*, 11(3): 353–359.

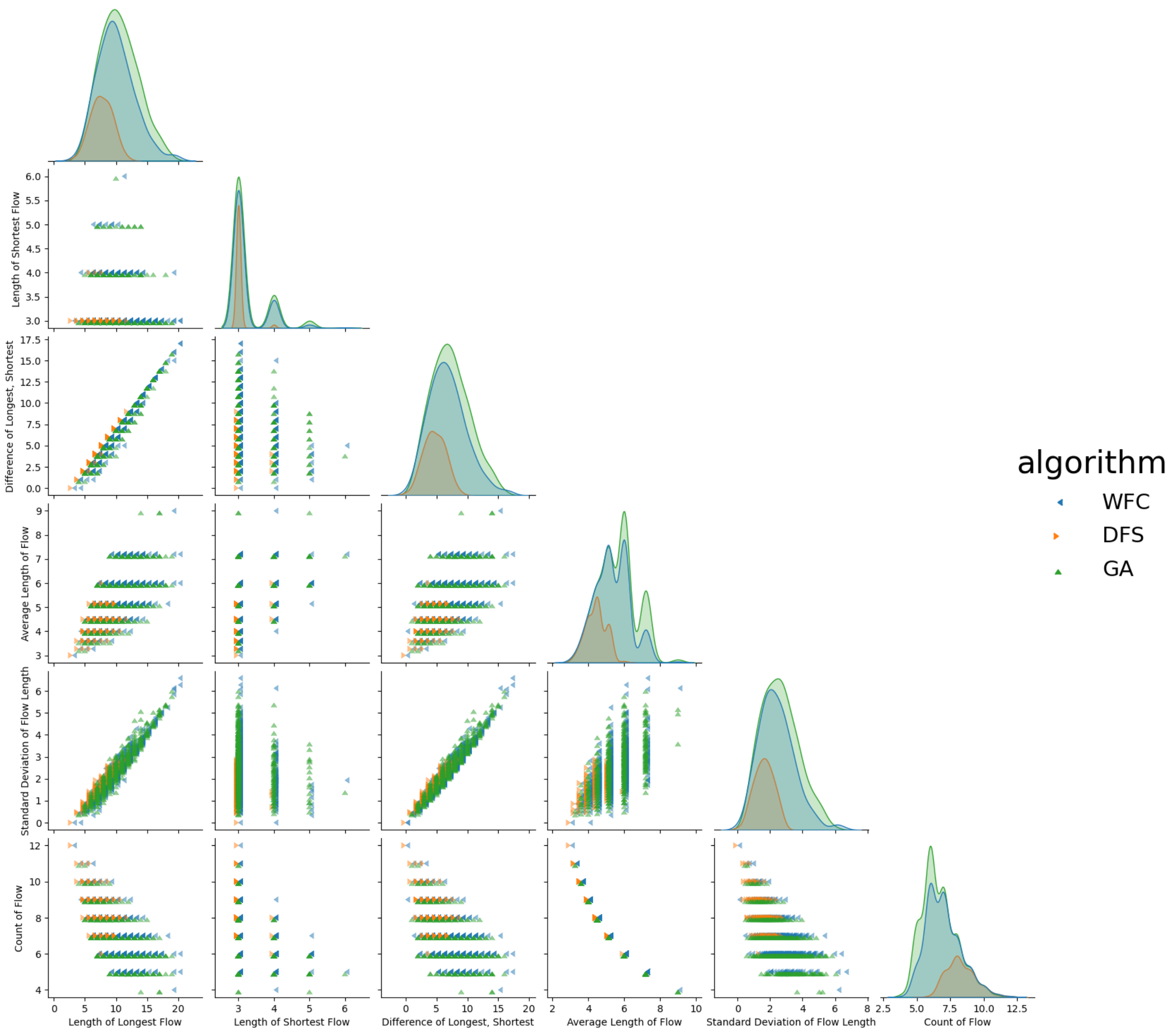


Fig. 17: Kernel Density Estimation corner plot: visualizing the statistical distribution of maps generated by WFC (blue), DFS (orange), and GA (green). Feature labels corresponds to those in Table VI. The diagonal graph is a uni-variate distribution plot is drawn to show the marginal distribution of the data in each column. The others represent graphs with two different variables on the x and y axes.

[6] Summerville, A.; Snodgrass, S.; Guzdial, M.; Holmgard, C.; Hoover, A. K.; Isaksen, A; and Togelius, J. 2018. "Procedural Content Generation via Machine Learning (PCGML)." *IEEE Transactions on Games*, 10(3): 257–270.

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

[8] Taylor, J.; and Parberry, I. 2011. "Procedural generation of sokoban levels." *In Proceedings of the International North American Conference on Intelligent Games and Simulation*, 5–12.

[9] Murase, Y.; Matsubara, H.; and Hiraga, Y. 1996. "Automatic making of sokoban problems." *In Pacific Rim International Conference on Artificial Intelligence*, 592–600. Springer.

[10] Ashlock, D. 2010. "Automatic generation of game elements via evolution." In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, 289–296. IEEE.

[11] Mantere, T.; and Koljonen, J. 2007. "Solving, rating and generating Sudoku puzzles with GA." In *2007 IEEE congress on evolutionary computation*, 1382–1389. IEEE.

[12] Shaker, M.; Sarhan, M. H.; Al Naameh, O.; Shaker, N.; and Togelius, J. 2013. "Automatic generation and analysis of physics-based puzzle games." In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, 1–8. IEEE.

[13] Fatemi, B.; Kazemi, S. M.; and Mehrasa, N. 2014. "Rating and generating Sudoku puzzles based on constraint satisfaction problems." *International Journal of Computer and Information Engineering*, 8(10): 1816–1821.

[14] Rychnovsky, T. 2014. "Procedural Generation of Puzzle Game Levels." [Online]. Available: https://bit.ly/3MxNEMh Accessed on: 2022-08-08

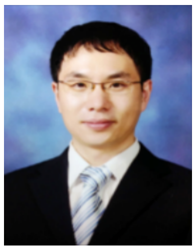
[15] Torrado, R. R., Khalifa, A., Green, M. C., Justesen, N., Risi, S. and Togelius, J. (2020, August). Bootstrapping conditional gans for video game level generation. In 2020 IEEE Conference on Games (CoG) (pp. 41-48). IEEE.

[16] Jain, R., Isaksen, A., Holmgård, C. and Togelius, J. (2016). Autoencoders for level generation, repair, and recognition. In Proceedings of

the ICCC workshop on computational creativity and games (Vol. 9).
[17] Gumin, M. 2016. "Wave Function Collapse" Github Repository. [Online]. Available: https://github.com/mxgmn/WaveFunctionCollapse Accessed on: 2022-08-08.
[18] Gumin, M. 2016. "Texture Synthesis" Github Repository. [Online]. Available: https://github.com/mxgmn/TextureSynthesis Accessed on: 2022-08-08.
[19] Gumin, M. 2016. "Conv Chain" Github Repository. [Online]. Available: https://github.com/mxgmn/ConvChain Accessed on: 2022-08-08.
[20] Freehold Games, L. 2015. "Caves of Qud" Steam. [Online]. Available: https://store.steampowered.com/app/333640/ Accessed on: 2022-08-08.
[21] Plausible Concept, O. S. 2018. "Bad North" Steam. [Online]. Available: https://store.steampowered.com/app/688420/ Accessed on: 2022-08-08
[22] Team, O. E. L.; Stooke, A.; Mahajan, A.; Barros, C.; Deck, C.; Bauer, J.; Sygnowski, J.; Trebacz, M.; Jaderberg, M.; Mathieu, M.; et al. 2021. "Open-ended learning leads to generally capable agents." *arXiv preprint arXiv:2107.12808.*
[23] Kim, H.; Lee, S.; Lee, H.; Hahn, T.; and Kang, S. 2019. "Automatic generation of game content using a graph-based wave function collapse algorithm." In *2019 IEEE Conference on Games (CoG)*, 1–4. IEEE.
[24] Kim, H.; Hahn, T.; Kim, S.; and Kang, S. 2020. "Graph Based Wave Function Collapse Algorithm for Procedural Content Generation in Games." *IEICE TRANSACTIONS on Information and Systems*, 103(8): 1901–1910.
[25] Mac, A.; and Perkins, D. 2021. "Wave Function Collapse Coloring: A New Heuristic for Fast Vertex Coloring." *arXiv preprint arXiv:2108.09329.*
[26] Merrell, P. C. 2009. Model synthesis (Doctoral dissertation, The University of North Carolina at Chapel Hill).
[27] Mackworth, A. K. 1977. Consistency in networks of relations. Artificial Intelligence 8, 1, 99–118.
[28] Mohr, R., and Henderson, T. C. 1986. Arc and path consistency revisited. Artificial Intelligence 28, 2, 225–233.
[29] Merrell, P. C. 2021. Comparing Model Synthesis and Wave Function Collapse.
[30] Softpedia. 2010. "Strimko 1.01." [Online]. Available: https://bit.ly/3T08I0E Accessed on: 2022-08-08.
[31] Veider, D. 2019. "Chain Sudoku - 200 Hard to Master Puzzles 8x8 (Volume 21)." *Independently Published.*
[32] Jr, E. P. 2010. "Beyond Sudoku." *The Mathematica Journal*, 11(3).
[33] Dudeney, H. 1917. Problem 252 – "A Puzzle for Motorists." *Amusements in mathematics.* [Online]. Available: https://bit.ly/3rVzhrH Accessed on: 2022-08-08
[34] Rosenblatt, M. 1956. "Remarks on some nonparametric estimates of a density function." The annals of mathematical statistics, 832-837.

Hwanhee Kim received a Bachelor's degree from the Department of Urban Planning & engineering from Yonsei University in 2007. From 2011, he joined Nexon Korea as a game designer. From 2017, he is working at NCsoft Korea as a senior technical game designer and also pursuing a M.S. degree from the Department of Games in Hongik University.

Beomjoo Seo received the B.S. and M.S. degrees from the Department of Computer Engineering, Seoul National University in 1994 and 1996, respectively, and the Ph.D. degree in Computer Science from the University of Southern California in 2008. He was formerly a Senior Research Fellow at the School of Computing, National University of Singapore. He is currently an assistant professor at the School of Games in Hongik University.



Shinjin Kang received an MS degree from the Department of Computer Science & Engineering from Korea University in 2003. After graduation, he joined Sony Computer Entertainment Korea (SCEK) as a game developer. From 2006, He has worked at NCsoft Korea as a lead game designer. He received Ph.D. degree in Computer Science & Engineering at Korea University in 2011. And he is now a professor at the School of Games in Hongik University.