Abstract

While working on a division of the Wave Function Collapse (WFC) algorithm into chunks, that would be generated by a player while exploring the game world, the following questions came to mind.

Firstly does a validation algorithm exist to check, if a given set of tiles and constraints will not create an empty super state?

Secondly if the WFC algorithm is divided into chunks, does the validation algorithm for the base algorithm need to be changed?

Contents

Chapter 1

Introduction

While the Wave Function Collapse (WFC) is intended to use an input bitmap image to create a larger version of this image, it can also be used to create game worlds. In this use case, the game developer defines a set of tiles and constraints on which the game world should be created (see Minecraft example [1]).

While trying out WFC algorithm for a 2d tile based world generation I asked myself the question if this algorithm could be used to create game world, while the player is exploring it.

For this to be done, the WFC algorithm needs to be divided into smaller problems. Each of these problems would be a smaller part of the world and all divided problems together would then represent the games world. Such approaches are known as chunk generation (see [1]).

While working on the division of the WFC algorithm into a chunk based generation algorithm the question arouse what should happen, if the algorithm couldn't produce a valid result. Since one must assume, that the player already interacted with the world, the algorithm can't be restarted to generate a new game world.

This means before a chunk based game world could be generated using the WFC algorithm, the tile and constraint set needs to be validated. The validation needs to check if for the given data a possible combination exists, that would result in a contradiction.

This document analysis the WFC algorithm to create such a validation algorithm and re-examines it for any needed changes, to be applicable to a chunk based version of the algorithm.

Chapter 2

Fundamentals

These are the fundamentals, of how the WFC algorithm can be used with a given tile set to create a 2d game world. All information is my understanding of the GitHub page of the WFC algorithm (see [3]) and its applications.

2.1 Wave Function Collapse (WFC)

To use the WFC algorithm to create a 2d tile based game world a set of tiles and corresponding constraints is needed. The constraint of each tile tells the algorithm, which tiles can neighbor it in the cardinal directions.

In the beginning of the algorithm every tile in the world will be a complete super state, which allows the placement of any tile. These super states can be seen as sets of all possible tiles, that could be placed at its position.

To start the world generation a random position is selected and the super state is collapsed to a single tile contained in its state set. Afterwards the surrounding super states need to be updated, to only allow tiles, that are part of the tile's constraint.

These updates will result in updates to their neighbors, creating the waves, that give the algorithm its name.

After the update waves are finished, a new super state will be selected to be collapsed. This position should be next to the previously collapsed tile.

Later on, if a super state is next to multiple tiles, its state will be the intersection of all constraints of the surrounding tiles.

2.1.1 Wave Propagation of the Collapse

The wave started by the first tile to be collapsed, will update every state, so that only the tiles are contained in it, that are part of any tile constraint of the previously updated super states. This update wave will be executed, till no super state will be changed anymore.

The figure 2.1 contains a visual representation of the propagation behavior, where s is the first tile that is collapsed at the start of the algorithm and i_1 is the first propagation, while i_2 to i_5 are the updated resulting from previous updates.

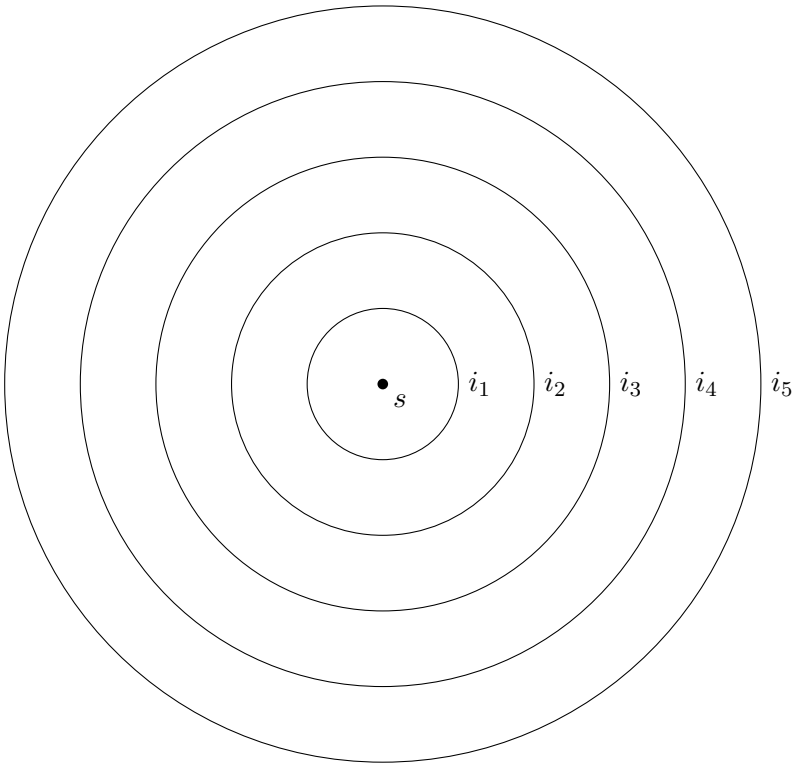


Figure 2.1: Wave Propagation Display

2.1.2 Abstraction of the Wave to Tiles

While the update propagation of the collapse can be seen as concentric waves, through the abstraction to tiles, the geometric form will be changed. The propagation will move along the cardinal directions. The propagation along these directions can be seen as cardinal axis, with their origin in the first tile. The update wave will update all states along the axis and the 45° lines connecting the axes, resulting in a concentric diamond pattern, as seen in figure 2.2.

By applying the concentric diamond wave to a tile grid, figure 2.3 can be created. This figure shows the update wave being executed in a tile based game world. Here s is the first tile to be collapsed in the world and i_1 to i_3 represent the concentric diamond waves.

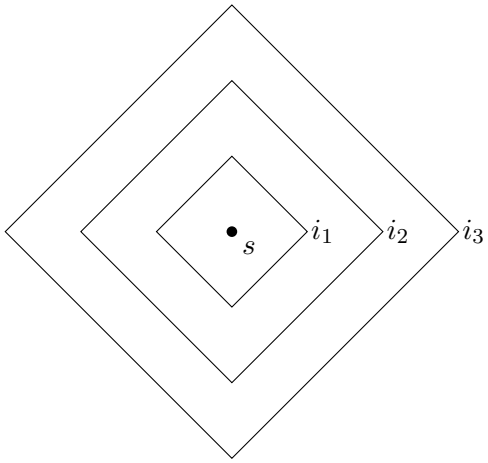


Figure 2.2: Wave Propagation Display with Diamonds

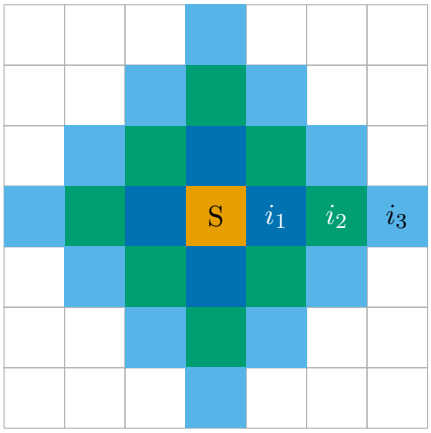


Figure 2.3: Example Tile Grid

2.2 Mathematical Definitions

To define a validation algorithm for a tile set and its constraints, a mathematical definition of these is needed. Since the used tiles and their constraints can be seen as sets the following definitions can be created.

Set of all possible Tiles

Every tile can be seen as a single numeric identifier, therefore the set of tiles can be defined as follows.

$$\begin{aligned} &1.\ V \subseteq \mathbb{N} \\ &2.\ V \neq \emptyset \end{aligned} \tag{2.1}$$

A Constraint

A constraint can be seen as the set of possible tiles allowed next to a tile. Furthermore, a constraint shouldn't be empty, since every tile needs to be able to have neighbors. Therefore, the constraint definition can be defined as follows.

$$\begin{aligned} &1.\ C \subseteq V \\ &2.\ C \neq \emptyset \end{aligned} \tag{2.2}$$

Tile Set with Constraints

Based on the previous definitions the tile set with the corresponding constraints for each tile can be defined as follows.

$$S = \{(v, C) \mid v \in V,\ C \subseteq V\} \tag{2.3}$$

2.3 Global Restriction

In most games there are tile combinations that aren't allowed to be created, for example two tiles of opposite natures shouldn't be generated next to one another.

This restriction can be defined with the following equation.

$$\exists (a, C_a), (b, C_b) \in S, a \neq b : C_a \cap C_b = \emptyset \tag{2.4}$$

Chapter 3

Derivation of a Validation Algorithm

To devise a validation algorithm for a set of tiles and their constraints it is needed to know what data needs to be validated for which properties. The equation 2.3 defines the input data to be validated. To know what the validation needs to check, the WFC algorithm needs to be analyzed.

3.1 Restricting the Wave Propagation

The biggest part of the validation algorithm, is the validation of no creation of empty super states, while the propagation wave after a collapse is executed. Looking at figure 2.3 and using the fundamental of any tile after the first one collapsing neighboring an already collapsed tile, the wave propagation can be restricted for a deterministic movement.

After the first tile is collapsed, all surrounding super states need to be updated (i_1 in 2.3). The next tile will be collapsed from these states. While this would normally result in a new update wave, this can be ignored for know, by collapsing all other tiles in the space of i_1 in 2.3. After all these tiles are collapsed, the WFC algorithm then can update all super states contained in the i_2 region of 2.3.

This changes the WFC algorithm from collapsing tiles and executing update wave propagations to a more iterative behavior. After the first tile is collapse, each concentric wave around will execute the following two steps, till the world border is reached.

1. Update all super states in the current ring
2. Collapse all super states in the current ring

3.2 Analysis of WFC Algorithm

After these changes are done, the algorithm must be analyzed for the following steps.

1. Collapse of the first tile
2. Wave i_1

3. Wave i_2
4. Wave i_3 and higher

Furthermore, the validation algorithm should also respect the restrictions written in section 2.3.

3.2.1 Collapse of the Starting Tile

In the beginning of the WFC algorithm a random complete super state, containing all possible tile ids, will be collapsed into a randomly selected tile.

For this to be achieved, the set of tile and constraint pairs must not be empty. This can be represented with the following equation.

$$|S| > 0 \tag{3.1}$$

3.2.2 First Wave Iteration

After the first tile is collapsed, the restricted algorithm will move to the first iteration of wave propagation as seen with i_1 of figure 2.3.

The first step in the wave iteration is to update the super states surrounding the first tile, to be the constraint of it. The second and last step of the iteration is to collapse all of these states into tiles.

For the collapse to create new tiles it is required, that the constraint corresponding to the starting tile isn't empty. This requirement can be defined with the following equation.

$$\forall (v, C) \in S : C \neq \emptyset \tag{3.2}$$

3.2.3 Second Wave Iteration

The second iteration of the wave propagation will update the super states surrounding the tiles of the previous iteration. For the updates in the cardinal directions the super states will be just the constraints of the previous tile, since no other tile is neighboring this position. Therefore, the validation of this step is equal to the first wave iteration.

The other super states will contain the intersection of the constraints of the surrounding tiles. The tiles used to update the super states can be referenced to as parent(s).

Looking back at figure 2.3 it can be seen, that each of the intersections will be calculated from two tiles. Both of these tiles stem from the constraint of the starting tile. Therefore, it can be said, that no intersection of constraint corresponding to the tiles of the starting tile can be the empty set.

Adding to this, that the intersection of identical sets is just the set itself (see [2]), the intersection validation can be reduced to different tile constraints.

Based on these observations the equation below can be derived to validate that no empty set will be introduced as a super state in the second iteration, regardless of which starting tile would be chosen.

$$\forall (s, C_s) \in S,\ \forall a, b \in C_s,\ a \neq b :\ C_a \cap C_b \neq \emptyset \tag{3.3}$$

3.2.4 Higher Wave Iterations

The analysis of the second wave iteration includes backtracking to the previous iterations, because the starting tile and its constraints are used to validate the updating of super states in the second iteration. Since this is the case, the next step is to analyze, if the previous statements are enough to validate all wave iteration after the second one or if additional information is needed.

Looking back at figure 2.3 it can be seen, that every tile in the later wave iterations will be formed by either the intersections of the constraints of the two parent tiles or the constraint of the single parent in case of the cardinal directions. This behavior is better seen in figure 3.1, since two zones are highlighted by black borders to showcase the behavior.

Zone z_1 show the previously analyzed case regarding the starting tile. On the other hand zone z_2 represents any constellation of tiles in the later iterations. The green tile in zone z_2 is the origin of all other tiles in the zone. The light blue tiles are results of the constraint of the origin tile, while the uncolored tile would be collapsed form the intersection of the constraints of the other two tiles.

This behavior directly overlaps with the previous case. Therefore, it can be said, that no extension of validation arguments is needed for wave iterations after the second one.

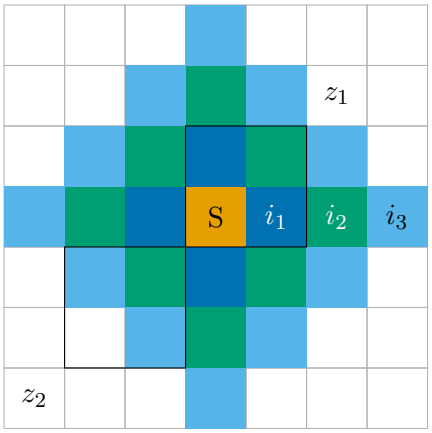


Figure 3.1: Example Tile Grid with Zones

3.2.5 Reflection on the Global Restriction

One last analysis that needs to be done for the validation algorithm is to check, if it adheres to the global restriction stated in section 2.3. The equation 2.4 in this section states, that the intersection of constraints of two different tiles can be empty. This equation stands in direct contradiction with the equation 3.3 for the validation of higher iterations of the wave propagation, since it says that the intersection of the constraints of two different tiles must not be empty.

While the contradiction stands from a mathematical point of view, the interpretation for the generation of game worlds solves it. When two tiles can't be next to one another, it is required to either define the tile set so, that they can't generate in proximity to one another. This case is handled via equation 3.3, since it would create an empty set and flag the given tile set as not valid. Another method to handle this situation in generating a game world, would be to introduce one or more tiles to bridge the space between these tiles.

3.2.6 Summarization

With the analysis of the WFC algorithm finished, the properties to validate against can be summarized as follows.

1. The set of tile and constraint pairs isn't empty (3.1)
2. The empty set is no constraint of any given tile (3.2)
3. The intersection of the constraints of two different tiles in any constraint is not empty (3.3)

3.3 Definition of the Validation Algorithm

To validate a given set of tiles and their constraint the algorithm first needs to check, that the given data structure is not empty. Afterwards the algorithm needs to iterate over every tile and constraint pair, to check if the data structure representing the constraint isn't empty. Lastly the algorithm needs to check every combination of constraints of distinct tiles for the result of their intersection. Since every tile has a corresponding constraint, which must be readable from the data structure using the tile ids, the algorithm can be build on validating pairs of tile ids. Furthermore, the number of pairs can be reduced, by applying the commutative property of the intersection operation, as shown in equation 3.4, for more information see [2]. Applying this property to the tile id pair, the equation 3.5 can be applied to the validation algorithm.

$$A \cap B = B \cap A \tag{3.4}$$

$$(a, b) = (b, a) \tag{3.5}$$

Since multiple constraint can result in the same pair of tile ids the algorithm should keep track of the found pairs. It should be noted, that the tracker needs to respect the commutative aspect of the validation algorithm shown in 3.5. Furthermore, it could be the case that all possible pairs are found before every constraint is checked for the tile pairs contained in it. The upper bound of possible pairs can be calculated by using the binomial coefficient.

The binomial coefficient can be used to calculate the number of different combinations of items in a set regardless of their order ([4]). Since the validation algorithm has a fixed amount of tiles, that will be taken from the tile set, to create the constraint intersection, and the order of the tiles is irrelevant, the binomial coefficient can be used to calculate the upper bound of the number of pairs, that could be found. Therefore, the following equation can be used to calculate the upper bound.

$$\binom{|S|}{2} = \frac{|S|!}{2!(|S|-2)!} \tag{3.6}$$

3.3.1 Example in Python

The validation algorithm can be broken down into a common and a part pair validation part. This split can be done, because the common part only validates that the input data follows the

mathematical definitions and restraints (2.2). A function for the pair validation would handle the validation of all possible tile pairs and their constraints intersections not being empty.

The python code seen in 3.3.1 shows the function handling the mathematical side, while the function in 3.3.1 handles the pair validation.

Both functions use a python logger object to create a log file, that will contain information about any none valid tile pairs or constraints found. Since the user would want all information about the none valid part of their tile set, the algorithm only sets a flag for the tile set to being valid or not, instead of directly returning false on a none valid state.

All example code can be found on GitHub under `https://github.com/PatrickSuszek/WFC-Tile-Set-Validation`.

```python
def _common(logger: Logger, data: dict[int, set[int]]) -> bool:
    valid = True

    # check the existence of tile constraint pairs
    if len(data) == 0:
        logger.info("The given constraint set is empty")
        valid = False

    # check that no constraint is empty
    for t, c in data.items():
        if len(c) == 0:
            logger.info(f"The constraint of tile {t} is an empty set")
            valid = False

    return valid
```

Code 3.1: Python Function Common Validation

```python
def two_dim(logger: Logger, data: dict[int, set[int]]) -> bool:
    if not _common(logger, data):
        return False

    # create helper variables
    valid = True
    found_pairs = []
    seen_pairs = set()
    max_pair_cnt = math.comb(len(data), 2)

    # find all possible pairs of tile ids
    for _, constraint in data.items():
        if len(found_pairs) == max_pair_cnt:
            break
        pairs = list(itertools.combinations(constraint, 2))
        for p in pairs:
            if p[0] != p[1]:
                fs = frozenset(p)
                if fs not in seen_pairs:
                    seen_pairs.add(fs)
                    found_pairs.append(p)

    # check all possible tile pairs
    for p in found_pairs:
        c_a = set(data[p[0]])
        c_b = set(data[p[1]])

        if len(c_a & c_b) == 0:
            logger.info(f"The intersection for the pair {p} is an empty set")
            valid = False

    return valid
```

Code 3.2: Python Function Pair Validation

3.3.2 Runtime and Space Estimation

The estimations of the used memory and basic operations to be done in the runtime, are based on the number of entries in the tile set and the upper bound of the number of possible tile pairs.

All estimations are done for python. The estimations can be applied to other programming languages, if the sets are hashed and the data structures have an attribute for their length.

3.3.2.1 Runtime Estimation

$$
\begin{aligned}
&1.\ \mathcal{O}(1)\\
&2.\ \mathcal{O}(|S|)\\
&3.\ \mathcal{O}(|S| * \binom{|S|}{2})\\
&4.\ \mathcal{O}(|S| * \binom{|S|}{2})\\
&5.\ \mathcal{O}(2 * |S| * \binom{|S|}{2} + |S| + 1) = \mathcal{O}(|S| * \binom{|S|}{2} + |S|)
\end{aligned}
\tag{3.7}
$$

1. Checking if the given dictionary is not empty is an atomic operation, since python saves the dictionary length in its attributes and the greater than operation is also atomic.

2. To check if no empty constraint is in the dictionary, each set size attribute must be checked. As stated above this check is atomic, but it must be executed for every dictionary entry.

3. For every tile in the tile set, the possible pairs needs to be calculated.

4. For each pair an intersection will be calculated. Each constraint could have all tiles in it, therefore the time to create the intersection is the number of tiles. If the used data structure isn't hashed, this would be squared, since every entry of the other set must be looked at for each value in a set.

5. The last Big O notation represents the full runtime of the algorithm.

3.3.2.2 Space Estimation

$$
\begin{aligned}
&1.\ \mathcal{O}(|S|^2 + |S|)\\
&2.\ \mathcal{O}(\binom{|S|}{2})
\end{aligned}
\tag{3.8}
$$

1. Each dictionary entry is a set of tile IDs and a single tile ID, where the set can be all tile IDs available.

2. Since the helper variables have either constant size or the same length as the found pair list, the used size for the pair validation can be defined as the upper bound of pairs.

Chapter 4

Reflection on Chunk Division

While the derived algorithm validates, that a given tile set doesn't result in an empty super state, while executing the WFC algorithm, the applied restrictions have nothing to do with a division into chunks.

Therefore, the next part of this document will analyze what changes the alteration of the WFC algorithm into a chunk based algorithm will introduce. This analysis will also reflect on the exploration behavior of players in a game world. Each chunk will only be generated, if either is near the chunk or if the player is in the chunk, depending on the implementation. Since it can never be known, how a player will move through the world, every possible combination should be seen as something that will happen.

4.1 World States in Chunk Based Generation

The start of a chunk based WFC algorithm version is equal to the start of the normal WFC algorithm. Every tile in every chunk will be set to the complete super state and a random chunk will be selected for the start. Afterwards the WFC algorithm will be executed normally, till the wave propagations reach the chunk borders, which can be seen as the world borders in the normal version.

For the generation of the other chunks, it will be assumed, that the chunk generation will be executed in a generation zone around the chunk housing the player. The population of the chunks will follow the same rules as the wave propagation. Meaning, that the following point will be adhered to for all chunks other than the starting chunk.

- Only chunks next to already generated chunks will be generated
- The chunks will be generated iteratively following the wave propagation

Based on those points, it can be seen, that the chunk generation will look like the tile generation seen in figure 2.3.

Since the player movement acts as a restriction to the chunk generation, it required to know every possible state for a chunk in its super state and its neighbors. Starting from these combinations, the validation algorithm can either be extended or validated for its completeness.

4.1.1 Chunk Combinations

The figure 4.1 shows the parts of a chunk grid that will be relevant for the analysis of possible chunk combinations in the generation. The chunk labeled α represents the chunk to be generated, while the chunks a - d are its influencing neighbors.

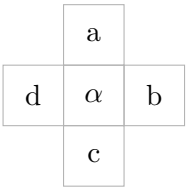


Figure 4.1: Example Chunk Grid

The surrounding neighbors will be either generated or be in the super state. Viewing the generated neighbors as a set, the power set of the set $\{a,\ b,\ c,\ d\}$ can be used to list all possible combination of generated chunks surrounding the chunk to be generated.

$$P(A) = \left\{ \begin{array}{l} \varnothing, \\ \{a\}, \{b\}, \{c\}, \{d\}, \\ \{a,b\}, \{a,d\}, \{b,c\}, \{c,d\}, \\ \{a,c\}, \{b,d\}, \\ \{a,b,c\}, \{a,b,d\}, \{a,c,d\}, \{b,c,d\}, \\ \{a,b,c,d\} \end{array} \right\} \tag{4.1}$$

Analyzing the sub-sets the following information can be derived.

1. The empty set isn't possible, since a chunk other than the starting chunk will always be generated to at least one generated chunk.
2. Only a single neighboring chunk is already generated.
3. Two neighboring chunks are already generated and they are adjacent to one another.
4. Two neighboring chunks are already generated and they are opposite to one another.
5. Three neighboring chunks are already generated and two of the three chunks are adjacent to the third one.
6. All four chunks are already generated.

4.2 Chunk Combination Implications

As previously stated, the chunk generation is just the application of the normal WFC algorithm. Instead of using the worlds boundaries as the end for the wave propagations, the chunk alteration lets the wave propagation end at the chunk boundary. Furthermore, every tile in a not generated chunk consists of the complete super state.

Normally the WFC algorithm would select a random tile to start the generation. Because the chunk has neighbors that are already generated, it needs to be analyzed how the algorithm may need to be changed, so that the newly generated chunk fits its neighbors. If this isn't done, the chunks would have no correlation to one another, which would result in the constraints of the tile set being broken.

4.2.1 Single Chunk

For the extension of the wave propagation from a single chunk into the new one, the tile to start the population of the chunk should be selected at the border of the two chunks, since this is where the previous wave ended.

Choosing a random super state on the border to be collapsed, can result in a new wave, that isn't extending the previous one. This newly created wave would partly overlap with the previous wave, but in the other part it would run opposite to it. This behavior is shown in the figure 4.2.

The blue part of the figure is the already generated chunk, where the first tile to be collapsed at the border is labeled α. Represented in black is the chunk to be generated with the starting tile β. The differently colored arrows depict the wave propagation directions relative to the first tile collapsed in the row. It can be seen, that the wave propagation in the new chunk, runs against the previous wave, till it reaches behind α.

Based on this analysis it can be derived, that for the correct propagation of the wave, the starting tile of the new chunk must be the neighbor of the first tile to be collapsed at the border of the parent chunk.

The position of this tile could either be calculated or saved. Since the player is exploring the world and a backtracking could be impossible, the position of the relevant tile should be saved as metadata of the chunk.

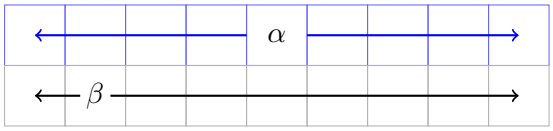


Figure 4.2: Example Random Tile, One Chunk

4.2.2 Adjacent Dual Chunks

The combination of two adjacent chunks to a chunk to be generated, can further be divided into two cases. The first category can be seen in figure 2.3, where this combination always results from a common parent chunk. Through the player's exploration of the world a combination of two adjacent chunks can be generated, that don't share a common parent. This can be seen in figure4.3, where the blue squares are chunks to be generated and the blue arrow is a possible exploration path of the player to generate this situation.

This is observation is important, since one possible solution, would be to start the chunk generation in the corner, where the two parent chunks meet. The corner position would then be generated using the normal wave propagation, since the surrounding tiles were created by using the constraint of the corner tile of the parent chunk from both chunks.

Since both cases need to be handled and the waves of both chunks need to be further propagated, the chunk based WFC algorithm should collapse a tile on each border and propagate two waves, to populate the chunk. Calculating the wave propagation pattern of this approach, results in figure 4.4. Here it can be seen, that there are two possibilities for the two waves to meet.

1. The constraints of tiles of both waves will be used to create the border, where the waves are overlapping (subfigures a and b)

2. If the two are only touching one another (subfigure c), no information exchange between the waves will be used to create the border

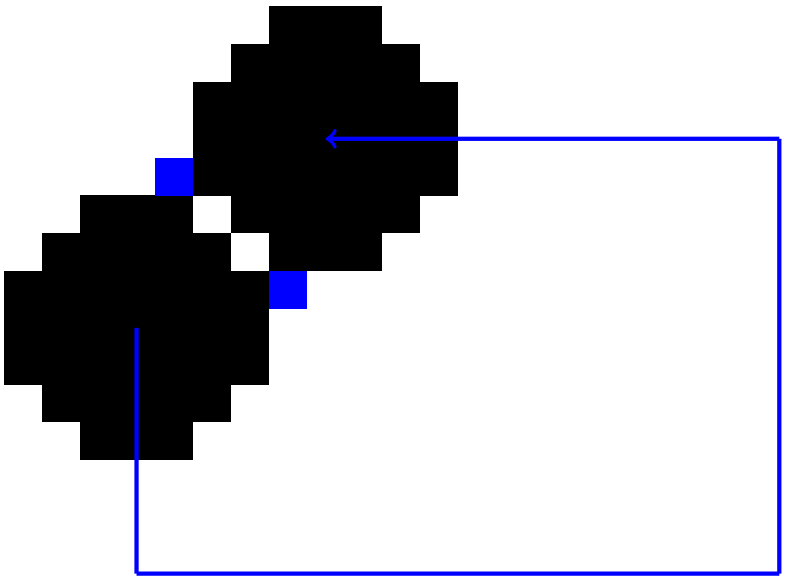
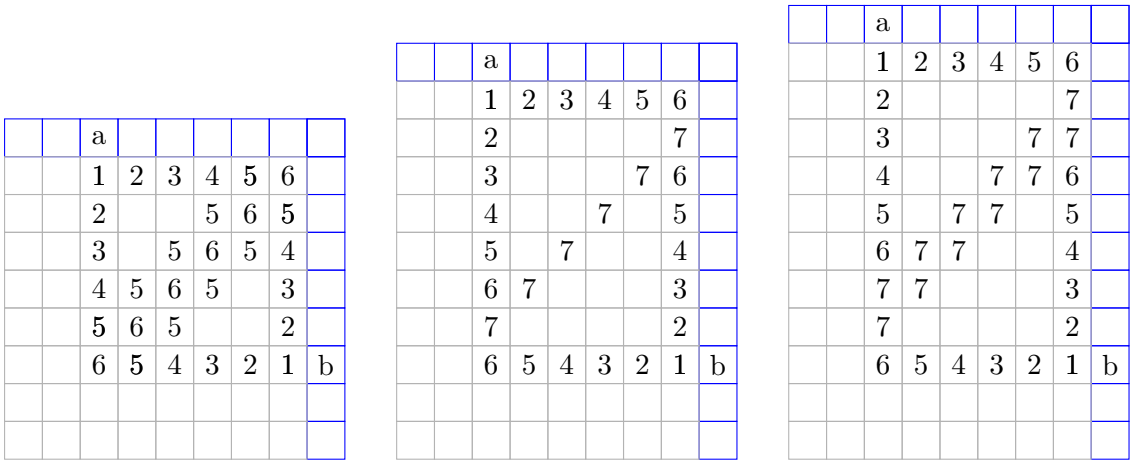


Figure 4.3: Example Adjacency Without Common Parent



(a) Equidistant And Overlapping

(b) Not Equidistant And Overlapping

(c) Not Equidistant And Touching

Figure 4.4: Different Wave Propagations For Two Adjacent Chunks

4.2.2.1 Overlapping Implications

As it can be seen in the subfigures a and b of the figure 4.4, there are three cases for the intersection of constraints to calculate the super states of the border.

1. The constraints of two tiles of each wave will be used for the intersection. (The diagonal)
2. The constraint of one tile of each wave will be used for the intersection. (The corner)

3. The constraint of one tile of one wave and two tiles of the other wave will be used for the intersection. (The side)

To validate this behavior the algorithm needs to execute the following verifications.

1. The intersection of any four tile constraints isn't empty
2. The intersection of any two tile constraints isn't empty
3. The intersection of any three tile constraints isn't empty

4.2.2.2 Touching Implications

Subfigure c of figure 4.4 represents the two waves touching, without any exchange of information. For the border to be valid in this case, the waves would need to propagate another iteration out and check, if the tiles generated by the other wave are valid for itself. A problem arises if this isn't the case, since the generation of the chunk would either need to revert to the iteration before the border was generated or the chunk needs to be completely reset.

This behavior shouldn't be part of the chunk based WFC algorithm, since it can't be validated, that a solution exists. Because of the global restriction (2.3) the intersection of two constraints can be empty if they shouldn't be next to one another. Adding to this the exploration behavior of the player it can't be determined which state a chunk would have. Therefore, it could be that no solution for this chunk exists. Going further along with the backtracking, this would mean, that previous chunks would need to be recalculated. This must generally not be done, because the player could already have interacted with the chunk.

Since the case of both waves touching to form a border can't be validated, this case shouldn't be part of the chunk based WFC algorithm.

4.2.2.3 No Touching Border Restriction

Taking another look at the wave propagation pattern in the subfigure a of figure 4.4 it can be seen, that an overlap of the two waves can be achieved if the two starting tiles are at the same relative position on their respective borders.

Since the starting tiles are selected using the position of the first tile to be collapsed at the parents border, this tile needs to always be at the same relative position. This can be achieved by restricting the chunk geometry and the starting behavior of the algorithm. By defining the chunks as squares with odd side length and starting the world generation in the middle of the starting chunk, the pattern in figure 4.5 will be achieved.

This pattern will always reach the tile at the borders of the starting chunk. Adding to this, that the generation of any following chunk will start at the border to continue the wave propagation, the overlapping pattern of subfigure a of figure 4.4 will be achieved.

Please note, that the square with an odd side length is only an example and every other geometry could be used, as long as it adheres to the listed properties. Furthermore, since the relative starting positions for the extension of the wave propagations will be always the same, the position of the first collapsed tile at each border of a chunk doesn't need to be saved to its metadata.

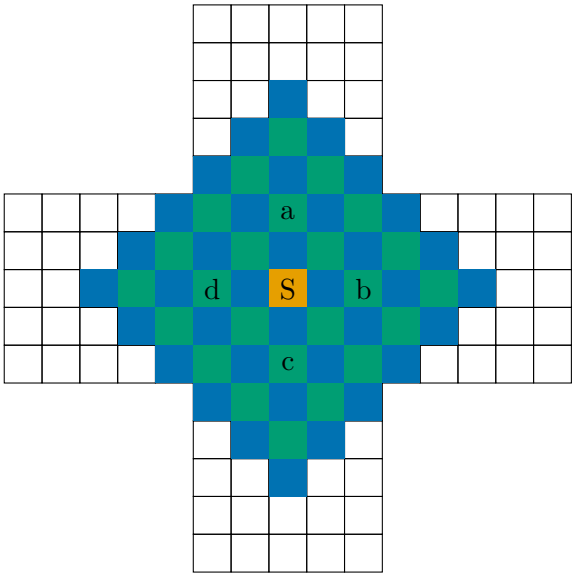


Figure 4.5: Example Chunk Definition, No Touching

4.2.3 Opposite Dual Chunks

Applying the new restrictions to the wave propagations of two opposite chunks figure 4.6 can be derived. Here it can be seen that the two waves will overlap at the height of the middle point of the chunk.

The super state for the middle tiles of the chunk will be calculated by intersecting the constraints of the uncorrelated tiles above and below it. Every other tile along the border will be collapsed from the intersection of three tiles.

Therefore, the validation for this chunk combination can be reduced to the following.

1. The intersection of any two tile constraints isn't empty
2. The intersection of any three tile constraints isn't empty

			a			
4			1			4
	4		2		4	5
		4	3	4	5	6
			4	**5**	**6**	7
		4	3	4	5	6
	4		2		4	5
4			1			4
			b			

Figure 4.6: Wave Propagation For Two Opposite Chunks

4.2.4 Three Chunks

By extending the wave propagation of the three chunks figure 4.7 can be derived.

Here it can be seen, that the chunks a and b are adjacent to chunk c. The meeting of the waves of chunks a and b, seen on the right side of the figure, are identical to the combination case of two opposite chunks. It can also be seen, that the meeting of the waves of the middle chunk c and the adjacent chunks a and b behave like the meeting of two adjacent chunks for their respective borders.

Only the middle tile of the chunk doesn't correspond to the cases of the combinations of two chunks, since its super state is calculated from the intersection of constraints from three independent tiles.

				a			
	4			1			4
		4		2		4	5
			4	3	4	5	6
c	1	2	3	**4**	**5**	**6**	7
			4	3	4	5	6
		4		2		4	5
	4			1			4
				b			

Figure 4.7: Wave Propagation For Three Chunks

4.2.5 Four Chunks

The extension of the wave propagation for the combination case of four chunks results in figure 4.8

Here it can be seen, that all four waves meet at an overlapping border, which will be calculated by using the information of two waves. This can therefore be reduced to the combination case of two adjacent chunks.

The middle tile of the chunk, will be collapsed from the intersection of the constraints of four independent tiles.

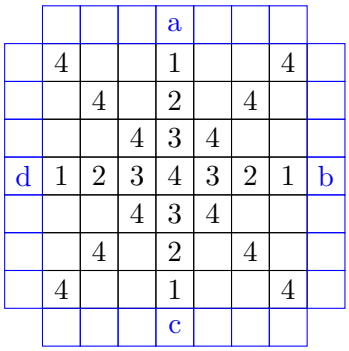


Figure 4.8: Wave Propagation For Four Chunks

4.2.6 Implication Summarization

The summarization of the derived information for the validation of a tile set for a chunk based WFC algorithm can be split into the restrictions for the algorithm and the validation steps.

4.2.6.1 Chunk Generation Restriction

- A chunk should ideally be a square
- The side length of the square should be an odd number
- The starting point of the world generation should be the middle point of the starting chunk

4.2.6.2 Validation Algorithm Extension

The validation algorithm needs to be extended for the following cases. Cases already included by previous combinations aren't listed.

Single Chunk: No extension needed.

Dual Adjacent Chunks: All constraint pairs need to have a none empty intersection.

$$\forall T \subset S,\ |T| = 2 : \quad \bigcap_{(a,C)\in T} C \neq \emptyset. \tag{4.2}$$

Dual Opposite Chunks: All constraint triples need to have a none empty intersection.

$$\forall T \subset S,\ |T| = 3 : \quad \bigcap_{(a,C)\in T} C \neq \emptyset. \tag{4.3}$$

Three Chunks: No further extension needed.

Four Chunks: All constraint quartets need to have a none empty intersection.

$$\forall T \subset S,\ |T| = 4 : \quad \bigcap_{(a,C)\in T} C \neq \emptyset. \tag{4.4}$$

4.3 Validation Algorithm Extension

Based on the analysis of the chunk division, the validation algorithm needs to check all possible intersections of two, three and four tile constraints. Using the properties of sets and the intersection operator, these validation steps can be reduced to the single case with the highest set count ([2]). This can be done, since if the intersection of any number of sets isn't empty, any given intersection of any part of these sets will also not be empty, since they contain at least one element that is the same.

The equation 4.5 showcases this property for the validation of the 2d chunks.

$$\begin{aligned} &Q = \{A, B, C, D\}, \quad A, B, C, D \subset \mathbb{N}. \\ &\bigcap_{X\in Q} X \neq \emptyset \implies \forall M \subset Q,\ |M| \in \{2, 3\} : \bigcap_{X\in M} X \neq \emptyset. \end{aligned} \tag{4.5}$$

4.3.1 Extended Algorithm Example in Python

The extended validation algorithm uses a new function to validate the chunks. In this function all possible tile combination of four will be created ant their constraint will be intersected. Analog to the previous validation function, this function logs the empty intersection tile combinations and only sets a validation flag, that is returned at the end.

```python
def chunk(logger: Logger, data: dict[int, set[int]], neighbors: int = 4) -> bool:
    if not _common(logger, data):
        return False

    # create helper variables
    valid = True

    # calculate all combinations of tiles
    tiles = set(data.keys())
    combinations = list(itertools.combinations(tiles, neighbors))

    for comb in combinations:
        constraints = []
        for c in comb:
            constraints.append(data[c])

        intersection = set.intersection(*constraints)

        if len(intersection) == 0:
            logger.info(f"The intersection for the combination {comb} is an empty set")
            valid = False

    return valid
```

Code 4.1: Python Function Chunk Validation

4.3.2 Extended Algorithm Runtime and Space Estimation

4.3.2.1 Runtime Estimation

$$
\begin{aligned}
&1.\ \mathcal{O}(1 + |S|)\\
&2.\ \mathcal{O}(|S|)\\
&3.\ \mathcal{O}(\binom{|S|}{4})\\
&4.\ \mathcal{O}(|S| * \binom{|S|}{4})\\
&5.\ \mathcal{O}(1 + 2 * |S| + \binom{|S|}{4})
\end{aligned}
\tag{4.6}
$$

1. The runtime of the _common function.
2. The time to create the set of tile ids.
3. The time to create all possible combinations for four neighbors
4. The time to execute the intersections of all combinations.
5. The last Big O notation represents the full runtime of the algorithm.

4.3.2.2 Space Estimation

$$
\begin{aligned}
&1.\ \mathcal{O}(|S|)\\
&2.\ \mathcal{O}(\binom{|S|}{4})\\
&3.\ \mathcal{O}(|S|)\\
&4.\ \mathcal{O}(|S| + \binom{|S|}{4})
\end{aligned}
\tag{4.7}
$$

1. Space used by the tile id set.
2. Space used to hold all tile id combinations for four neighbors.
3. Space used to hold all information about constraints and the intersection.
4. Total space used.

Chapter 5

User Errors and Edge Cases

While the extended validation algorithm can validate a tile set for its validity in a chunk based WFC algorithm, it doesn't handle any edge cases or input errors from the user.

5.1 User Errors

The user defined data can either have tiles, that aren't in any constraint or it can have tile ids in constraints, that aren't part of the tile set.

The first case isn't a problem, since this tile could only be selected as a starting tile and its constraint would just be handled like normal.

The second case would create an error, since the tile doesn't exist. Therefore, the validation algorithm should be stopped at this point.

Another error would be the multi definition of a tile id and its constraint. This can be avoided by using a hashed data structure, that uses the tile ids as its keys.

5.1.1 Code Changes

To handle the input error of using tile ids, that aren't part of the tile set, the common part of the validation algorithm should be updated. Since the corresponding check needs the set of available tile ids, which is also used by the chunk validation function, this data object is also returned.

```python
def _common(logger: Logger, data: dict[int, set[int]]) -> tuple[bool, set[int]]:
    valid = True
    # check the existence of tile constraint pairs
    if len(data) == 0:
        logger.info("The given constraint set is empty")
        valid = False
    # check that no constraint is empty
    for t, c in data.items():
        if len(c) == 0:
            logger.info(f"The constraint of tile {t} is an empty set")
            valid = False

    # check only known tiles are used
    tiles_known = set(data.keys())
    tiles_used = set.union(*data.values())
    valid &= tiles_used.issubset(tiles_known)

    if not valid:
        logger.info(f"Constraints are using unknown tiles")

    return valid, tiles_known
```

Code 5.1: Python Validation Handling User Error

Since the return of the _common function has changed the following changes need to be applied to the other validation functions.

```python
# old
if not common(logger, data):
    return False
valid = True

# new
valid, _ = _common(logger, data)
if not valid:
    return False
```

Code 5.2: Python Validation Update User Error

5.2 Edge Cases

The current implementation of the validation algorithm for chunks can't validate any tile set with less than four tiles.

5.2.1 Code Changes

To enable the chunk function to validate tile sets with less than four tiles, the selection count can be set to the minimum of either neighbor count or the tile set size.

```python
def chunk(logger: Logger, data: dict[int, set[int]], neighbors: int = 4) -> bool:
    valid, _ = _common(logger, data)
    if not valid:
        return False

    # edge case handling, tile count less then neigbors
    neighbors = min(neighbors, len(data))

    # calculate all combinations of tiles
    tiles = set(data.keys())
    combinations = list(itertools.combinations(tiles, neighbors))

    for comb in combinations:
        constraints = []
        for c in comb:
            constraints.append(data[c])

        intersection = set.intersection(*constraints)

        if len(intersection) == 0:
            logger.info(f"The intersection for the combination {comb} is an empty set")
            valid = False

    return valid
```

Code 5.3: Python Validation Handling Edge Cases

Chapter 6

Conclusion

Going into the analysis of the Wave Function Collapse algorithm there where two questions that this document wanted to answer.

1. Can a given tile set be checked if it will generate no empty super state?
2. Does the validation algorithm need to be adjusted for a chunk based variant and if so how must it be adjusted?

By restricting the WFC algorithm to finish generating every concentric ring of the wave propagation, before moving to the next ring, a validation algorithm could be deducted.

Looking at all possible generated and not generated chunk combination around a chunk to be generated and how the wave propagation would then behave in the new chunk it could be deducted, that the validation algorithm needed to be adjusted. Going into a deeper analysis of the behaviors, when propagating the generating wave through the different chunks into the new chunk, the needed adjustments were clarified and the algorithm was updated.

While both validation processes are based on the tiles and chunks being square, the derived validation steps are valid for any geometric forms fitting the introduced restrictions.

It is furthermore possible to even extend the validation into more than two dimensions for the chunk generation, since this would only update the maximum number of tiles / cubes used for the generation of the middle tile / cube of the chunk.

6.1 Personal Note

When I started looking into this problem I couldn't find any sources containing an answer to any of my questions. Finishing the compilation of my analysis notes, I hope that this document while prove to be at least a helping hand to anyone having the same or similar questions as mine.

Acronym Overview

WFC Wave Function Collapse

Bibliography

[1] *Chunk*. `https://minecraft.wiki/w/Chunk`. minecraft.wiki.

[2] Herbert B. Enderton. *ELEMENTS OF SET THEORY*. `https://docs.ufpr.br/~hoefel/ensino/CM304_CompleMat_PE3/livros/Enderton_Elementsofsettheory_(1977).pdf`. ACADEMIC PRESS, INC. (LONDON) LTD., 1977. ISBN: 0-12-238440-7.

[3] Maxim Gumin. https://github.com/mxgmn/WaveFunctionCollapse.

[4] Bruce Ikenaga. *Binomial Coefficients*. `https://sites.millersville.edu/bikenaga/number-theory/binomial-coefficients/binomial-coefficients.pdf`. 2019.

List of Figures

List of source codes