Bachelorarbeit

Florian Bettin

Solving the Tantrix board game puzzle using a template-based Wave Function Collapse approach

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Florian Bettin

Solving the Tantrix board game puzzle using a template-based Wave Function Collapse approach

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Informatik Technischer Systeme
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Philipp Jenke
Zweitgutachter: Prof. Dr. Christian Lins

Eingereicht am: 18. Juni 2024

Florian Bettin

Thema der Arbeit

Lösen des Tantrix-Brettspielpuzzles mit einem vorlagenbasierten Wave-Function-Collapse-Ansatz

Stichworte

Wave Function Collapse, WFC, Tantrix, Hexagon, Template, prozedurale Generierung, PCG

Kurzzusammenfassung

Diese Bachelorarbeit wendet den Wave Function Collapse Algorithmus auf das *Tantrix* Brettspiel an, um automatisch Ausgabebilder zu erzeugen, die visuell den Regeln des Spiels folgen. Um dies zu erreichen, werden zunächst digitale Versionen der echten Spielsteine erzeugt, zusammen mit einer Datei, welche die erlaubten Nachbarschaftsbeziehungen der Steine beschreibt. Da die Spielsteine eine hexagonale Form haben, wird ein schon vorhandenes Framework dahingehend erweitert, dass es den WFC auch auf diese Form anwenden kann. Des Weiteren wird eine neue Funktionalität eingeführt, bei der ein Benutzer ein Bild mit einer handgemalten Form in das Programm laden kann. Der Algorithmus rekonstruiert diese Form dann mit den digitalen Steinen und füllt übrige Lücken im Bild automatisch auf. Am Ende dieser Arbeit wird die modifizierte Version des WFC Algorithmus hinsichtlich Laufzeit und Erfolgsrate evaluiert.

Florian Bettin

Title of the paper

Solving the Tantrix board game puzzle using a template-based Wave Function Collapse approach

Keywords

Wave Function Collapse, WFC, Tantrix, hexagon, template, procedural content generation, PCG

Abstract

This bachelor thesis applies the Wave Function Collapse algorithm (WFC) to the *Tantrix* board game in order to automatically create output images that visually adhere to the game's rules. To achieve this, digital representations of the game's physical tiles are created, along with a file describing the adjacency rules for the tiles. Since the *Tantrix* tiles are of hexagonal shape, a preexisting framework containing a basic implementation of the WFC is expanded to support hexagonal tiles. Furthermore, an additional functionality is added to the algorithm: A user can provide an image with a hand drawn shape to the program, which is then recreated by the algorithm using the digital *Tantrix* tiles. Any open spaces are afterwards filled in using the normal WFC. At the end of this thesis, the performance of the modified WFC algorithm is evaluated in terms of runtime and success rate.

Contents

List of Figures

1 Introduction

1.1 Motivation

Figure 1.1 shows an exemplary layout of the tiles from the *Tantrix* board game [1], which was first released in 1988. The rule book to this game features several ways to play. However, all variants share one common premise: Every tile that is placed on the table has to fit to all the neighbors surrounding it. This means that all colored lines that directly connect to each other show the same color. While most of the rules describe multiplayer games, there is also one interesting solitaire version explained: The player is motivated to take as many tiles as they wish and form different shapes with a certain color using these tiles (e.g. only paying attention to the yellow lines for the shape).

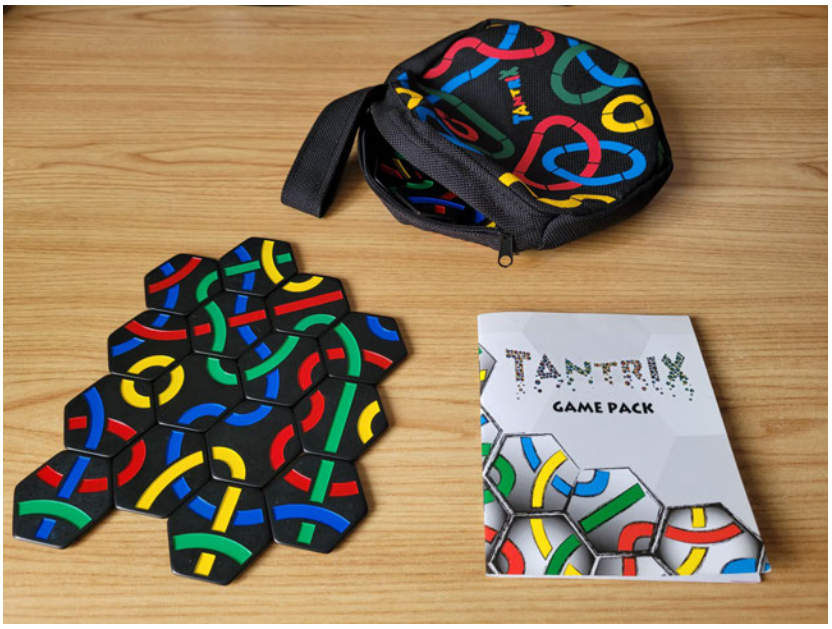


Figure 1.1: Example layout of *Tantrix* tiles next to the game package and the rule book.

[1]Website of the *Tantrix* board game. http://www.tantrix.com/. Accessed 06/13/24

In the field of procedural content generation (PCG), the wave function collapse algorithm (WFC), designed by Maxim Gumin (Gumin [2022]), gained widespread recognition after its release in 2016. In its basic form, the algorithm has two modes that it can work in. Either the user provides the algorithm with a small example image. Then the algorithm looks for patterns in this image and constructs a new, bigger image following the style of the example image. Or the user provides multiple small images of tiles coupled with adjacency constraints that define which tiles can be neighbors to each other. The algorithm then creates a big image by placing multiple tiles next to each other, following these adjacency rules. This mode is called the *Simple Tiled Mode.*

The *Tantrix* board game and the *Simple Tiled Mode* of the WFC both share the same premise of only placing tiles next to each other, if no adjacency rules are violated. This sparked the idea for this thesis: The physical tiles of the *Tantrix* game should be represented by digital versions, including a file that describes which tiles can be adjacent to each other. Those digital tiles can then be used in the WFC to automatically create output images with multiple tiles placed adjacent.
Since the supervisor for this thesis already provides a framework that includes a basic implementation of the WFC[2], this work should use the framework as a foundation. However, the WFC in the framework can only work with rectangular tiles, but the *Tantrix* tiles are hexagonal. Therefore, the framework has to be expanded in order to support hexagonal tiles.
Running the normal WFC with hexagonal tiles shall henceforth be called the *normal mode.* A second mode that shall be implemented is the *template mode.* This mode is inspired by the solitaire version for *Tantrix*, mentioned above. A user should be provided with an image of a grid of empty hexagonal tiles. In this image, the user can then draw an arbitrary shape, as long as the shape is one connected loop, in the end. The algorithm should then load this template image and recreate the given shape as best as possible by placing *Tantrix* tiles on a new grid. The rest of the grid can then be filled out using the normal WFC.

1.2 Overview

Apart from this introductory chapter, this thesis comprises six additional chapters. The second chapter explains important theoretical concepts that are the basis for the WFC and the implementation of this work. Chapter three discusses a related work whose authors also modified the basic WFC and measured the impact of those changes on the runtime and memory

[2] *CG Algorithms Datastructures framework.* https://git.haw-hamburg.de/Philipp.Jenke/cg_algorithms_datastructures. Accessed 06/07/24

usage of the algorithm. In chapter four, the concept for the software is outlined, detailing requirements which the program should meet, the architecture of the software, the general development model, and ideas for the implementation of the various functionalities that the program needs to provide. How exactly these functionalities are realized is then described in chapter 5. The performance of the implementation is evaluated in chapter 6, which is followed by the last chapter that summarizes this work and gives an outlook for possible future work.

2 Theory

This chapter first gives a short introduction to the general field of procedural content generation. Afterwards, three different techniques are discussed that were influential in the development of the Wave Function Collapse algorithm (WFC). Furthermore, the WFC itself is described in this chapter. Lastly, fundamental aspects for working with hexagons are introduced.

2.1 Procedural Content Generation

As its name already implies, procedural content generation (PCG), in its broadest sense, means that some form of artifact is automatically created by an algorithm. The kind of content that is being synthesized can be manifold and depends on the method that is applied. For example, the result could be a piece of music, an image/art, 3D models, or text (Angelides and Agius [2014]).
PCG is also an integral part in game development. Because it is tedious for the developer to model a whole game world by hand, PCG can be helpful in creating either the whole world on its own, or parts of it (Shaker et al. [2016]). For example, Lindenmaier systems (aka L-systems) can be used to build realistically looking trees and shape grammars are well suited for generating buildings.
In the game *No Man's Sky*[1], a whole universe is procedurally generated, where every planet that is visible to the player can also be visited. In the looter-shooter *Borderlands*, all the weapons that can be found are procedurally generated [2], as well.
It is apparent, that PCG can therefore facilitate the work of a game designer. However, if a developer wants to have more control over the creation process, a mixed initiative strategy can be chosen. There, e.g. while designing a game map, the designer would provide parts of the map themselves and the algorithm could take over at that point.
An additional benefit of PCG can be to save space during distribution of a game. The content that is being generated does not have to be created in advance and saved in files but can be

[1]No Man's Sky homepage. https://www.nomanssky.com/press/. Accessed 03/17/2024.
[2]Wiki for the Borderlands game. https://borderlands.fandom.com/wiki/Borderlands_Weapons. Accessed 03/17/2024.

generated on the computer where the game is played on (Togelius et al. [2011]).
Furthermore, especially in game genres such as rogue-likes [3], where a part of the game loop is to restart the game from the beginning frequently, procedural content generation can provide replayability to the game, because each runthrough is most likely different to the previous ones.
The rise of machine learning in recent years has also impacted PCG, with neural networks being able to synthesize content similar to the one generated by traditional methods (Summerville et al. [2017], Mitra et al. [2019]).

2.2 Texture Synthesis

The basic idea of texture synthesis is to create an output image that resembles an input image provided by the user. The input image is oftentimes smaller than the output image and is a means to provide the style that the output image shall have. In addition to creating a bigger image, texture synthesis can also be used to fill in holes in an image as they might appear, for example, after multiple images were merged while creating a panorama image.
In this context, *texture* means some kind of pattern in 2D. Typically, textures are categorized on a spectrum between *regular* textures and *stochastic* textures. While regular textures exhibit more structured patterns, e.g. an image of a stonewall, stochastic textures appear random, e.g. a roughcast.
There exist different approaches to realize texture synthesis, but this section concentrates on the method described by Efros and Leung [1999], because this was influential in the development of the WFC (Gumin [2022]).
Nevertheless, all methods have common goals they try to achieve (Wei et al. [2009]):

1. The output image should have dimensions as specified by the user.
2. The output should be similar to the input.
3. There should not be artifacts like visible seams, apparent tiling or misalignments in the output.
4. There should be no repetition of the exact same pattern in the output.

In the approach from Efros and Leung [1999], the output image is synthesized one pixel per time. For a new pixel p, for which a value has to be set, its surrounding pixels (which already

[3]Heise website addressing rogue-likes. https://www.heise.de/download/specials/Die-besten-Roguelike-Spiele-fuer-PC-Mac-7478489. Accessed 03/17/2024.

have values) are considered. They form the *context* of p and are taken from a squared window in whose center lies p and whose size is chosen by the user via a parameter.
For now, assume that all pixels in the output image are already set, but the one pixel p. To find a value for p, the algorithm will try to identify windows in the input image that are similar to p's context. To do this, a distance function d is used whose return value expresses the similarity between the window and p's context. For the function d, the sum of squared differences is used, combined with a Gaussian kernel. This punishes differences between the pixels that are in the vicinity of p more than pixels that lie further away. As a result, the local structure of the texture is better preserved than if all errors would be weighted the same.
It is unlikely that an exact copy of the context of p can be found in the input image. Therefore, at first, a context is identified that resembles p's context the most. Then further contexts, for which d does not exceed a chosen threshold, are searched.
The values of the pixels at the centers of all found contexts can then be assembled in a histogram. Finally, the value for p can be taken from this histogram, either by taking a random value from it or by weighting the choice according to the previous results from d.
Of course, in reality not all pixels but one will be synthesized, at the beginning. This can be solved by initiating the output image with a random 3-by-3 seed from the input image. The algorithm can then grow layer by layer from that seed. If the input only contains a hole that is to be filled, the algorithm can grow from the hole's edges. All pixels in the context of p that do not have a value yet can simply be ignored during the calculation of d.
The size of the context, which is chosen by the user, determines the appearance of the output image. A small size will often result in a more random looking output because potential patterns in the input might not be captured in the context.

2.3 Model Synthesis

Texture synthesis is able to create new, larger images that are similar in style to an input image. However, the algorithms from texture synthesis are mainly developed with 2D textures in mind. Model synthesis can be viewed as an extension of texture synthesis operating in the 3D space (note: theoretically it can also operate in 2D, but the main application is in 3D). It also works with an example input and produces a larger, similar output. But, in contrast to texture synthesis, the input does not consist of a 2D image but a 3D model. The output is a 3D model as well.
Model synthesis was originally devised by Paul Merrell in the 2000s. He has since refined the method, but Gumin took inspiration for the WFC in the earlier versions of model synthesis,

or more specifically, in the method that Merrell calls *discrete model synthesis*. This section therefore focuses on discrete model synthesis as presented in Merrell [2007].
Model synthesis is described by Merrell as a *general-purpose procedural modeling tool*, because the models that can be provided by the user as input only have to follow two rules:

1. They have to be on a 3D grid.
2. They have to be divisible into smaller pieces.

Therefore, as long as the user follows these rules, they can provide any desired model and thereby produce a variety of outputs.
Unlike in grammar based algorithms, where the user has to describe the rules that govern the creation process, in model synthesis these rules are automatically learned from the input model in the form of adjacency constraints.
Figure 2.1(a) shows a 3D model that could function as the input for the model synthesis

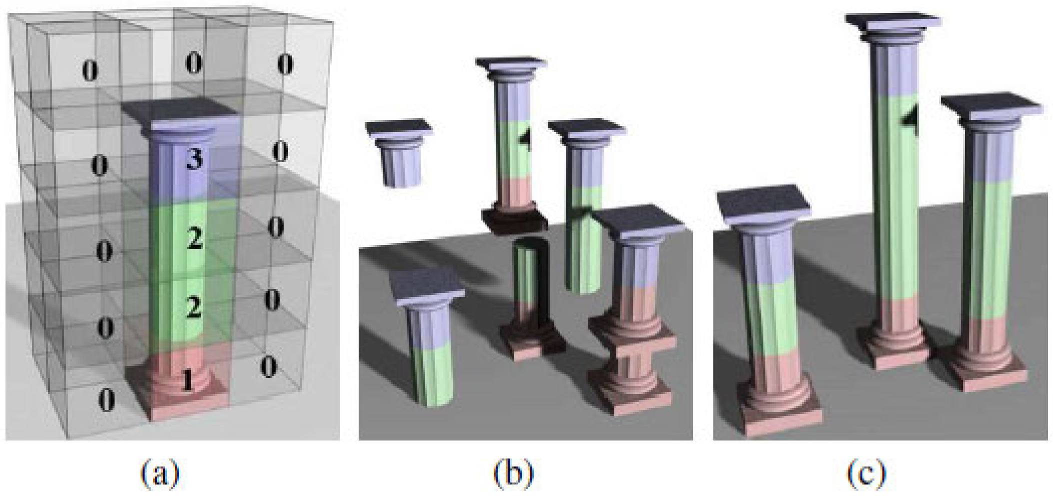


Figure 2.1: (a) Example model that could be an input, (b) Model that breaks adjacency constraints, (c) Model that follows adjacency constraints. Image taken from Merrell [2007]

algorithm. The pillar can be divided into four pieces that each fill exactly one cell in the 3D grid: The bottom part of the pillar, the middle part which consists of two identical pieces and the top part. Each distinct piece is assigned a label (1 through 3). Empty spaces are labeled 0. Figure 2.1(b) displays a generated model in which the different pieces from (a) are randomly placed in the 3D space. This model ignores adjacency rules that are expressed in (a). For example, a piece with label 3 always has to be on top of a piece with label 2. This rule is not followed in (b), which results in a top part of the pillar that is floating in the air. The model shown in (c) follows all adjacency constraints. This results in a model that is similar to the input. A model which adheres to all adjacency rules is called *consistent*.

The goal of model synthesis is to assign each space in the grid a label while still keeping

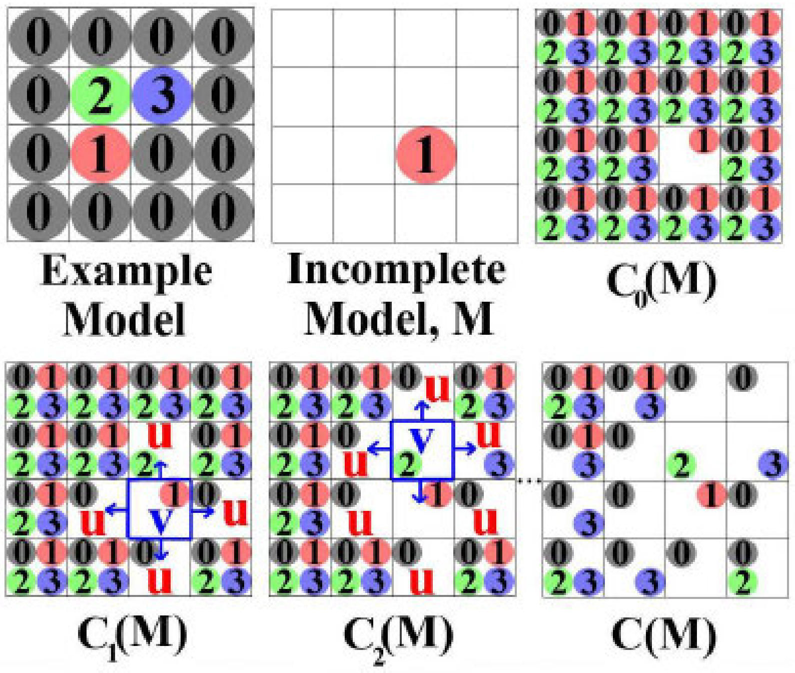


Figure 2.2: Example of a part of the model synthesis algorithm in 2D. Image taken from Merrell [2007]

the resulting model consistent. Figure 2.2 partly shows how the algorithm would work in 2D. C(M) is a list that, for each space of the grid, contains all the labels that could potentially be assigned to that space. At the start, every space could be assigned every label. The algorithm then randomly selects a space and, also randomly, chooses a label for that space. The effect of this selection on C(M) can be seen in $C_0(M)$. In order to follow the adjacency rules from the example model, the possible labels for the spaces bordering the just selected space have to be restricted, now. For those spaces, not all labels are possible anymore. This can be seen in $C_1(M)$. In sequence, for every space marked with a U, its neighbors also have to be restricted further. This causes a ripple effect propagating through the grid, which is why this step is

called *propagation*. Once propagation is finished, most of the spaces will not have all the labels available to them anymore. This is shown in C(M). At this point, the algorithm chooses the next space that still has more than one label available and starts the process again. This is repeated until every space is assigned exactly one label.
One problem of model synthesis is that creating a consistent model becomes harder, the larger the output model is supposed to be. Merrell solved this issue by dividing the grid into smaller parts and only solving one part at a time.
Figure 2.3 demonstrates this process. First, the whole model is initiated in a state that is known

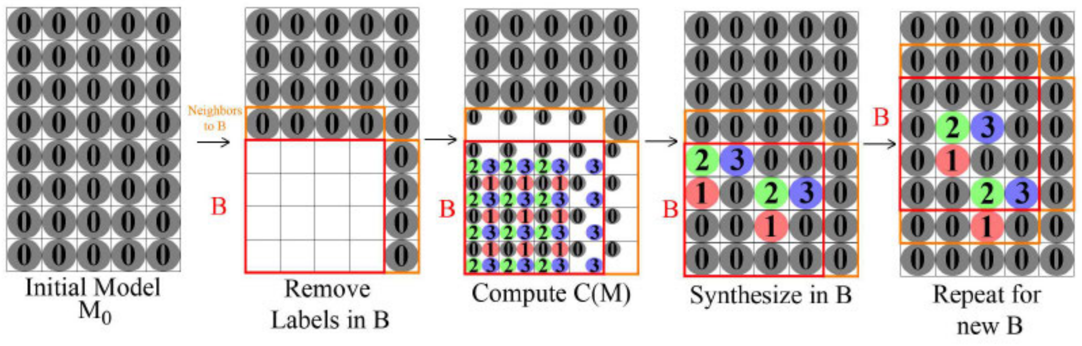


Figure 2.3: Model synthesis algorithm working on smaller parts of the grid. Image taken from Merrell [2007]

to be consistent (in this case the label 0 is assigned to each space). Then, a subarea of the grid is chosen to be worked on (named B). In the beginning, most spaces in B could be assigned every possible label, as was already seen in Figure 2.2. An exception to this are the spaces at the edges of B. Those spaces already have to take into account the neighboring spaces that are just outside of B (the areas marked with orange). Therefore, those edge spaces might not have all the labels available to them. Then, the algorithm can be carried out as described earlier. Once B is completely solved, a new subarea is chosen with the same dimensions as B previously, but moved one row or column further. All labels in that new area are removed and the process can be repeated. Over time, this will create a consistent model.
Even though the algorithm is explained for the 2D space here, it works for 3D just as well.

2.4 Constraint Programming

Constraint programming is a programming paradigm that emerged in the 1980s (Jaffar and Lassez [1987]). It is a generalization of logic programming with its well known programming languages like *PROLOG*. Therefore, predicate logic, which is at the heart of logic programming, is also an essential part of constraint programming.

One of the features of constraint programming, that separates it from other programming paradigms, is the declarative description of the problem that is to be solved rather than the imperative specification of the program. Constraint programming opens the possibility to the programmer of defining *what* the problem is instead of *how* the solution to the problem should be reached.

A typical application for constraint programming is solving combinational problems, like e.g. creating timetables, or optimization.

These are examples for the two major kinds of problems that constraint programming is used for: Constraint satisfaction problems (CSPs) and constraint optimization problems, which will be shortly introduced in the following. (Apt [2003])

2.4.1 Constraint Satisfaction Problems (CSPs)

A well known example for a CSP is the game of Sudoku [4]. There is a set of decision variables that have to be assigned a number between one and nine, each. In Sudoku, these variables are represented by the individual fields on the game board. Each decision variable has a set of values belonging to it - the *domain* of that variable. This domain contains all values that could be potentially assigned to the variable.

In addition, a CSP also contains a set of constraints on the decision variables that have to be fulfilled in a valid solution. In the case of Sudoku, these constraints are as follows:

1. Each row must contain the numbers one through nine, without duplicates.
2. Each column must contain the numbers one through nine, without duplicates.
3. Each block must contain the numbers one through nine, without duplicates.

A solution to this problem would be an assignment of one value to each decision variable, where none of the constraints is violated. A programmer would provide these constraints and the initial domains of the decision variables. The search for a solution is then handled by a constraint solver.

[4]Sudoku rules. https://www.sudokuonline.io/tips/sudoku-rules. Accessed 03/06/2024.

Constraint solvers contain a library of tests and operations that can be applied to different constraints in order to check if the problem can be solved at all (satisfiability) and if so, to calculate a solution. Different constraint solvers are limited to specific classes of constraints, i.e. the algorithms contained in them work very differently.
After being given the problem description, a constraint solver first tries to reduce the search space of the problem. This is the space of all possible variable assignments (which are not necessarily all valid solutions at this point).
The first tool for achieving this is to check *node consistency*. Node consistency is given, when all domains of the decision variables adhere to the unary constraints given in the problem. For example:

$$C_1 = (y \geq 3) \wedge (x > y) \wedge (x \in \{1, 3, 5, 7\}) \wedge (y \in \{2, 4, 6, 8\})$$

is not node consistent, because the value 2 from the domain of y breaks the unary constraint $(y \geq 3)$. The constraint solver would then remove that value from the domain to get

$$C_2 = (y \geq 3) \wedge (x > y) \wedge (x \in \{1, 3, 5, 7\}) \wedge (y \in \{4, 6, 8\})$$

which is node consistent (Hofstedt and Wolf [2007]).
A second way of pruning the search space is to consider *arc consistency*. Here, each binary constraint (involving two decision variables) contained in a problem is evaluated in sequence. For each variable that is part of that constraint, each element in its domain must have at least one partner in the domain of the other variable with which it fulfills the constraint (Mackworth [1977]). C_2 is not arc consistent, because e.g. the value 1 from the domain of x does not have a suitable partner in the domain of y that fulfills the constraint $(x > y)$. Removing the problematic values from the domains eventually leads to

$$C_3 = (y \geq 3) \wedge (x > y) \wedge (x \in \{5, 7\}) \wedge (y \in \{4, 6\})$$

which is arc consistent (Hofstedt and Wolf [2007]). A sophisticated algorithm for establishing arc consistency is the AC-4 algorithm developed by Mohr and Henderson [1985], which is also used in Maxim Gumin's implementation of the WFC algorithm (Gumin [2022]). These consistency checks do not necessarily create a solvable CSP. However, if at one point any of the domains do not have any values in them anymore, it becomes clear that the problem is not solvable.
After (potentially) reducing the search space, the constraint solver still has to find a viable solution to the given problem. This can be achieved by applying different search algorithms

to the search space such as the *branch and bound* algorithm. *Backtracking* is also often used during search. When the solver settles on a value for a variable, this information is propagated throughout the search space, which can eliminate further values from the domains of the other variables (Rossi et al. [2008]).

2.4.2 Constraint Optimization Problems

Constraint optimization problems are in essence also constraint satisfaction problems. Additionally, they feature an objective function that expresses the quality of a discovered solution. A famous example for this kind of problem is the Traveling Salesman Problem (TSP). In it, a graph is given, which is made up of a set of vertices accompanied by the edges that connect two of the vertices, each. Furthermore, each edge is assigned a numerical value which represents the cost of said edge. The objective of the problem is then to find a route through this graph, starting at a specific vertex, which includes each of the vertices exactly once. The cost of the solution is calculated by adding the costs of all the edges that are being used in this solution. This cost shall be minimized. Thus, a solution that has a lower cost than another one is considered "better" and therefore preferable (Gutin and Punnen [2007]).
Even though the objective function in the TSP should be minimized, this is not the case for all constraint optimization problems. The goal could be to maximize the objective function, just the same. In general, a constraint optimization problem consists of finding an assignment to the decision variables such that all constraints are satisfied and the objective is optimized (Rossi et al. [2008]).

2.5 Wave Function Collapse Algorithm

The Wave Function Collapse algorithm was originally released in 2016 by Maxim Gumin [5]. It quickly gained popularity, because it is easy to use, even for users that are not technically inclined, and the generation process is pleasing to look at.
As Gumin himself states (Gumin [2022]), the algorithm takes inspiration from texture synthesis and also from Merrell's discrete model synthesis (Merrell [2007]). Furthermore, the algorithm is deeply rooted in the field of constraint programming.
The indie skateboard game Proc Skater 2016 [6] was one of the earliest games to use the algorithm for procedural level generation. Since then, it has been been used in further games from different

[5]Gumin's original tweet first mentioning the WFC. https://x.com/ExUtumno/status/781833475884277760?s=20. Accessed 04/01/24.
[6]Proc Skater 2016. https://arcadia-clojure.itch.io/proc-skater-2016. Accessed 04/01/2024.

genres like in the strategy game Bad North [7], the town modeling game Townscaper [8] or the traditional roguelike Caves of Qud [9]. But the WFC is also used outside of the video game genre, for example as a generator for poems in the style of different authors [10].

The WFC has two modes that it can operate in: The *simple tiled mode* and the *overlapping mode*. These two modes dictate what the input to the algorithm looks like. In the overlapping mode, the user provides one input image that represents the style that the output image shall take on. The algorithm extracts different visual patterns from that input image together with the adjacency rules for those patterns.

In the simple tiled mode, the user does not provide one large image but rather multiple smaller images that represent the tiles with which the algorithm assembles the output image. The adjacency constraints for those tiles have to be provided by the user in a separate file.

Aside from the first step of handling the input, the algorithm works the same for both modes: The output image is instantiated in a completely unobserved state. This means that every NxN area (or cell, if the tiled mode is chosen) of the output can potentially be assigned any one of the input patterns. At the beginning, the algorithm chooses one of those areas at random since every area has the same options of patterns that it can be filled in with. This metric is called *entropy*. The entropy of an area becomes smaller, when less options of patterns to fill the area with remain. Aside from the beginning, the next area to place a pattern in is always determined by finding the area with the lowest entropy. Once an area is settled on, one of the possible patterns is chosen at random to fill the area with. The possibility for a pattern to be chosen depends on the input and is rarely uniform. Gumin calls this part of the algorithm (finding the next area and choosing a pattern) *observation*. When a pattern is chosen, the area (or cell) is said to be *collapsed*.

The next step is the *propagation*. Once a pattern for an area is settled on, this new information has to be spread to every neighbor of this area. The reason being, that the choice of the pattern will most likely reduce the options of patterns left for the neighbors, that can be placed in those areas without breaking adjacency constraints. If the options for at least one neighbor changed, this change in turn has to be propagated to the neighbor's neighbors. This process will continue until every area is updated.

Next, a new *observation* cycle is started followed by *propagation*. This is repeated until either the whole output image is completely filled out, or until a contradiction is encountered. A

[7]Strategy game Bad North. https://www.badnorth.com/. Accessed 04/01/2024.
[8]Modeling game Townscaper. https://www.townscapergame.com/. Accessed 04/01/2024.
[9]Roguelike Caves of Qud. https://www.cavesofqud.com/. Accessed 04/01/2024.
[10]Oisín: Wave Function Collapse for poetry. https://github.com/mewo2/oisin. Accessed 04/01/2024.

contradiction occurs when there is no valid pattern for an area that can be chosen without breaking adjacency constraints.

2.6 Hexagons

As can be seen in figure 2.4, hexagons can have two types of orientation: *Flat-top* orientation and *pointy-top* orientation. There is no fundamental difference between both orientations, but the code has to be adapted to the chosen orientation. Therefore, for simplicity, this work only uses hexagons in flat-top orientation.
Figure 2.4 also displays further properties of the hexagons that are being used[11]:

- The hexagons are of *regular* type, meaning that all sides have the same length.
- The $size$ of a hexagon is defined to be the distance from the middle point to a corner.
- The $height$ of a hexagon in flat-top orientation is $\sqrt(3) * size$.
- The $width$ of a hexagon in flat-top orientation is $2 * size$.
- A hexagon that is rotated by multiples of 60° yields a congruent hexagon.

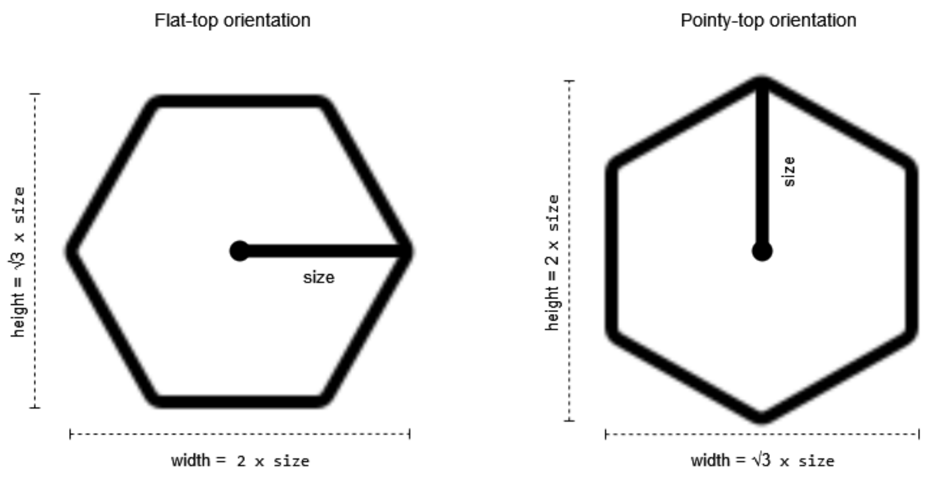


Figure 2.4: Two types of orientations for a hexagon with corresponding properties. Image inspired by https://www.redblobgames.com/grids/hexagons/#basics. Accessed 03/25/2024.

[11]Amit Patel's website about hexagons. https://www.redblobgames.com/grids/hexagons/#basics. Accessed 03/25/2024.

Figure 2.5 shows one way to combine multiple hexagonal tiles into a grid[12]. This grid uses a cube coordinate system, which consists of three dimensions q, r and s. The origin of this grid is located at the center of the grid where $q = r = s = 0$. This is because the maximal tile count in each dimension is chosen to be equal, here. It also causes the grid to be symmetric. In a cube coordinate system, the constraint

$$q + r + s = 0$$

always has to be satisfied. Therefore, moving from one tile to a neighboring tile involves increasing the value of one coordinate and decreasing the value of another one.
The horizontal distance between the centers of two neighboring hexagons is

$$distance_h = (3/2) * size$$

The vertical distance between the centers of two neighboring hexagons is

$$distance_v = \sqrt{(3)} * size$$

[12]See footnote 11.

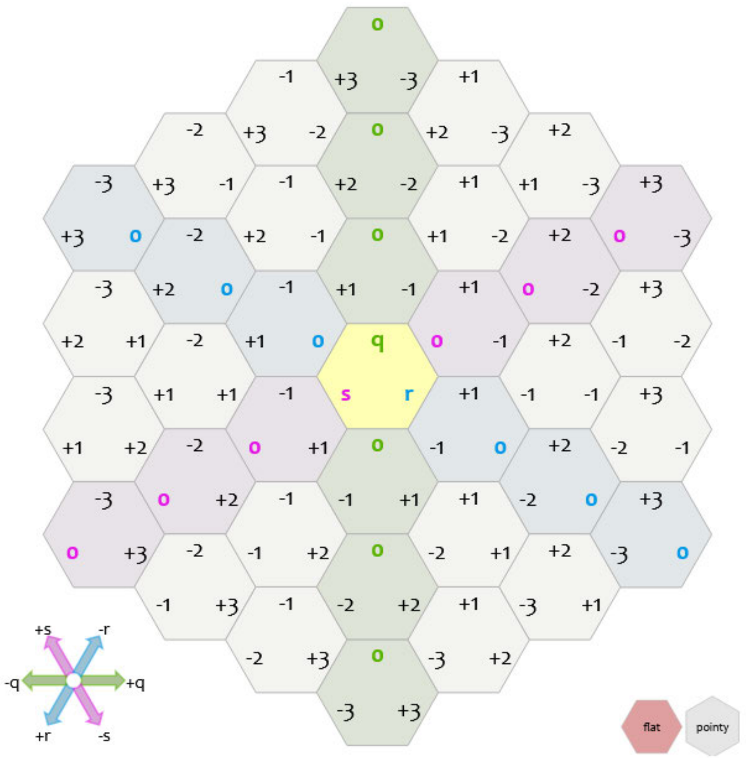


Figure 2.5: Hexagonal grid using cube coordinates. Image taken from https://www.redblobgames.com/grids/hexagons/#basics. Accessed 03/25/2024. Permission for use granted by creator.

3 Related Work

This chapter gives an overview of another work that introduces global constraints to the WFC and evaluates the impact of those constraints on the algorithm's performance.

3.1 Enhancing Wave Function Collapse with Design-Level Constraints

In their paper, Sandhu et al. [2019] describe different ideas to introduce design-level constraints to the WFC and the impact that those introductions have on the performance regarding computational time and memory usage.

In the introduction, the authors describe the benefits that PCG can provide in the process of game development, such as the acceleration of level creation. They also describe possible costs that reside in different PCG techniques. For example, constraint satisfaction solvers have an upfront cost that lies in designing the system. The programmers usually need to have a good understanding of the problem domain in order to formulate useful constraints. However, the payoff of this work are assets that can look more convincing and less random than creations that were generated by stochastic algorithms. Nevertheless, according to the authors, designers often shy away from said upfront cost and rather select a stochastic technique. In this context, the appearance of the WFC is noteworthy, because WFC is able to combine the advantages of constraint satisfaction solvers with the ease of use of other PCG techniques.

The authors also relate their work to the two games *Bad North* and *Caves of Qud*, because those games use *online* PCG, meaning the creation of the assets is not just done upfront (which would be *offline* PCG), but also during gameplay. Furthermore, both games use design oriented variations of the WFC: *Bad North* applies a new search heuristic to guarantee a walkable path in the game world and *Caves of Qud* uses a multipass system to create a more interesting game world. This inspired the authors to work on online, design focused modifications and extensions to the WFC, themselves.

After reiterating the general procedure of the WFC, the authors describe their alterations to the base implementation of the WFC. The first addition they call *weighted choice*. While the

base WFC uses entropy to decide which cell should be collapsed next, the tile that is placed in that cell is chosen at random. At this point, the weight that each tile contributed to the entropy value does not have an impact on the random choice of the tile, anymore. The authors state that they use "a combination of binary search with cumulative weight method" to also consider the weight of a tile during selection. However, they do not further describe this approach.
The first design constraints that the authors introduce are *non-local constraints*. For this, they modify the normal observation/propagation loop by adding an additional observation step. Furthermore, they introduce new non-tile items (like chests or keys) that can be placed by the WFC on top of normal tiles.
In the new observation step, a check is performed, if a forced observation of an item or a tile has to happen. For example, for a chest item, the designer can specify an upper and lower bound for the distance from the chest, where a key item has to be placed. Furthermore, a forced observation can also be triggered for a certain tile. Then, the algorithm places that tile in a subarea which is defined by the upper and lower bounds and subsequently observes the whole subarea by running a small scale WFC on that area.
Additionally, the authors establish a frequency counter that can limit the occurrence of items.
Another constraint that is introduced to the WFC is *weight recalculation*. This is achieved by adding a new trigger to the observation step. If the trigger condition is met, the weight for a specific tile is changed and the whole wave is updated accordingly. For example, the placement of a certain item in the map could trigger the recalculation of the weight of a tile to influence the look of the map, afterwards.
The last addition to the WFC is *area propagation*. Conceptually, it is similar to the *non-local constraints*. However, this method alters the propagation step of the original WFC. For a specified area, instead of propagating which objects/tiles are still allowed in that area, the algorithm spreads information about what objects/tiles are banned. For example, if a torch is placed in a room, then the tiles that make up the room are only allowed to be lightened ones.

In order to evaluate the performance of the different constraints, the authors use execution time and memory usage as metrics for *weighted choice*, *non-local constraints* and *weight recalculation*. Only one constraint is used per time and compared against the base implementation of the WFC. Each test consists of 50 runs using different map sizes of 10x10, 20x20, 30x30, 40x40, 50x50, and 100x100 tiles. Because *area propagation* causes a lot of conflicts during map creation, for this constraint the rate of successful creations out of 50 runs is used as a metric, instead.

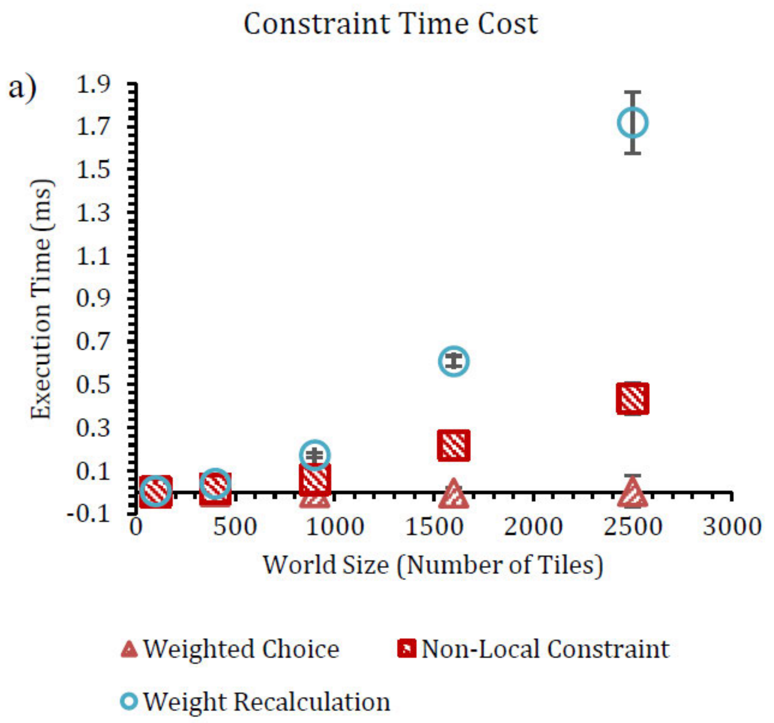


Figure 3.1: Time cost for different constraints and map sizes. Note: The authors mislabeled the units of measurement on the y-axis. It should be *seconds* instead of *milliseconds*. This becomes obvious when reading the paper. (Sandhu et al. [2019])

Figure 3.1 shows the results in respect to execution time. The *weight recalculation* constraint has the most impact on the runtime. For a map size of 2500 tiles, this constraint can delay the execution by up to 1.8 seconds. The *non-local* constraint has a less severe effect on the runtime, although it can still cause a delay of about 500ms compared to the baseline WFC. *Weighted choice* has the least negative influence. For 2500 tiles, the greatest delay is about 100ms. In contrast, the authors state that for 10000 tiles this constraint even decreases runtime by 3 seconds, on average.

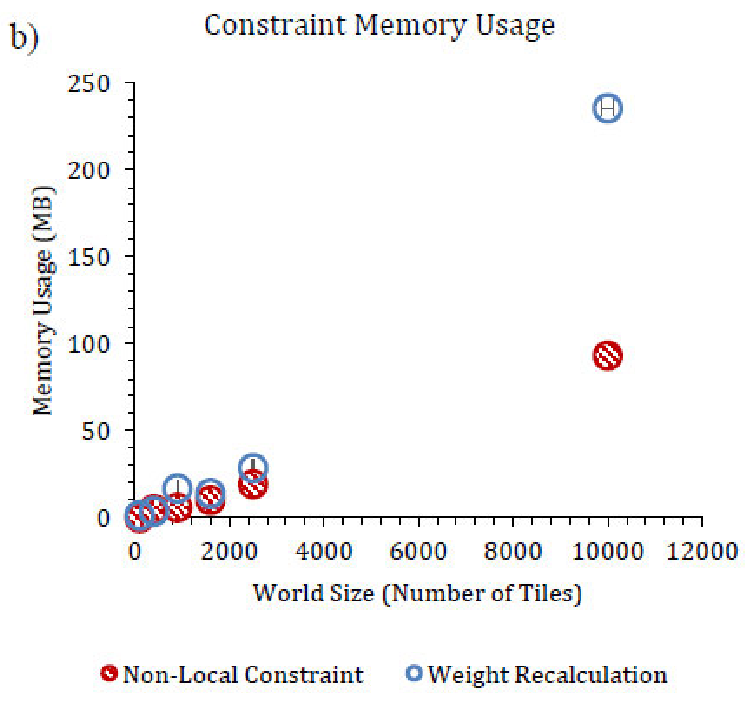


Figure 3.2: Memory usage for different constraints and map sizes (Sandhu et al. [2019]).

Figure 3.2 shows the average memory usage for different constraints and map sizes. It is noteworthy to point out two weaknesses in this figure: First, the authors did not include measurements for the *weighted choice* constraint. Second, the measurements are absolute values and there are no baseline measurements for the normal WFC. Therefore, it is unclear in what way the memory usage of the baseline WFC is impacted by the constraints. However, it still becomes clear that a growing map size increases the memory usage for both constraints. The results for the *area propagation* constraint are hardly discussed in the paper. The authors reference a table, but fail to actually include the table in the paper. However, they mention that the conflict rate for this constraint is around 60% for a map size of 100 tiles.

Even though parts of the evaluation of the constraints' impact on the performance is imprecise, the inclusion of the runtime measurements is still valuable to this thesis. The impact of introduced constraints on the execution time is considered most interesting in this work.

4 Concept

4.1 Problem

Before the WFC could be applied to the *Tantrix* board game puzzle, several preparations had to be done in advance. First, a digital representation of the real world *Tantrix* tiles had to be created. Next, the constraints describing what tiles can be adjacent to each other had to be formulated and saved in a file. Furthermore, the framework that this project is working with only supports squared tiles. Therefore, one major task was to extend the framework so that it is able to work with hexagonal tiles as well.
Once the normal WFC was able to correctly create images with hexagonal tiles, the software had to be expanded to be able to provide a template file, that contains an empty grid with hexagonal tiles, to the user. Then the functionality of processing the template file, which contains the hand drawn shape from a user, needed to be incorporated into the WFC.
Finally, for ease of use of the software, a simple GUI had to be created.

4.2 Requirements

4.2.1 Functional Requirements

Output

1. **General:**
   a) The output image shows hexagonal tiles. (Req.1)
   b) Each tile shows three lines. (Req.2)
   c) The lines on a tile have different colors. (Req.3)
   d) Possible colors for the lines are red, blue, green, yellow. (Req.4)
   e) Each line connects two edges on a tile. (Req.5)
   f) An edge cannot have more than one line crossing it. (Req.6)

   g) There are no contradictions: Two edges of adjacent tiles, that are touching, need to have a line of the same color crossing them. (Req.7)

   h) The grid on the output image has the same size as specified by the user. (Req.8)

   i) If the WFC is unable to generate a correct output, it is restarted until a valid output is created. (Req.9)

2. **Template mode:** In addition to the general requirements listed above, the output from the *template mode* also needs to meet the following requirements:

   a) The shape that is formed by the yellow lines resembles the hand drawn shape given by the user. (Req.10)

   b) Yellow lines are only used to recreate the shape given by the user. (Req.11)

GUI

1. The user can define the grid size. (Req.12)
2. The user can start the *normal mode* via a button. (Req.13)
3. The user can start the creation of an empty template file via a button. (Req.14)
4. The user can load the drawn in template file into the program via a button. (Req.15)
5. The user can start the *template mode* via a button. (Req.16)
6. If there was no image uploaded by the user before starting the *template* mode, an error is displayed on the screen. (Req.17)
7. If the grid size of the template file provided by the user does not match the currently selected grid size, an error message is displayed on the screen. (Req.18)
8. Once the algorithm is finished, the output is displayed on the screen. (Req.19)

4.2.2 Non-Functional Requirements

The program must meet the following non-functional requirements:

Organizational Constraints

- The implementation is based on the framework *CG Algorithms Datastructures*[1]. (Req.20)

[1] *CG Algorithms Datastructures framework.* https://git.haw-hamburg.de/Philipp.Jenke/cg_algorithms_datastructures. Accessed 06/07/24

Documentation

- Every method and class has to have *Javadoc* comments. (Req.21)
- Source code needs to be commented for better understanding. (Req.22)

Readability

- The naming in the source code has to be meaningful and descriptive. (Req.23)
- The naming has to follow the naming conventions introduced by Google [2]. (Req.24)
- For better understanding, the code follows the decomposition paradigm (Pohl and Rupp [2015]). (Req.25)

Testability

- The correct functionality of the program has to be verified via tests. (Req.26)
- The need for tests at a later stage is already kept in mind while designing methods. (Req.27)

Performance

- The algorithm finishes in a reasonable time, depending on the grid size. (Req.28)

4.3 Assumptions

In order for the algorithm to work, specifically for the *template mode*, there are some assumptions that need to be made concerning the hand drawn shape provided by the user:

- The user needs to draw the shape in red (RGB: 255/0/0).
- The hand drawn shape has to be closed (i.e. a loop).
- While drawing, the user can enter and leave a tile only once with the drawn line.
- The start and end of the shape has to be on the same tile. However, the end does not need to exactly hit the pixel of the start.

[2] Google style guide. https://google.github.io/styleguide/javaguide.html#s2.1-file-name. Accessed 05/20/2024.

4.4 Development Model

For the realization of the project, the incremental development model was used (Alshamrani and Bahattab [2015]). More specifically, the staged delivery model, as seen in 4.1. With this model, the project could be divided into smaller problems, namely the digitization of the hexagonal tiles and their constraints, the implementation of the *normal mode*, the realization of the *template mode* and lastly the implementation of a simple GUI. Furthermore, the model was flexible enough to react to possible suggestions and/or problems that might arise during development.

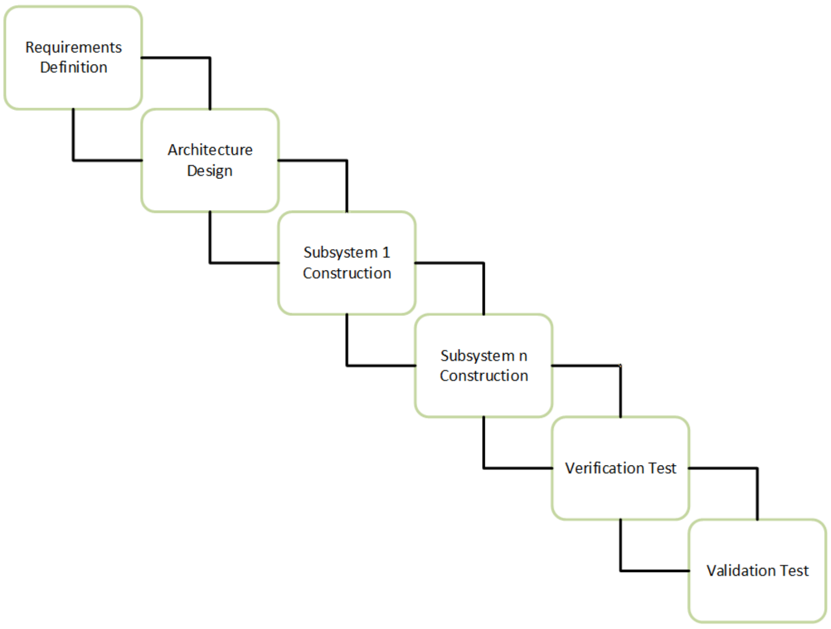


Figure 4.1: The staged delivery model. Image inspired by https://www.geeksforgeeks.org/software-engineering-incremental-process-model/. Accessed 05/19/2024.

4.5 Tiles

The physical board game version of *Tantrix* comes with 56 distinct tiles. An image of these real tiles can be seen in figure 4.2. Each tile shows three lines that each connect two edges of a tile. If a line connects two edges adjacent to each other, it shall be called *corner*. If it connects two edges that have one edge between them, it is a *curve*. Lastly, a line that connects two edges opposite from each other is called a *straight*.
These lines can appear in four different patterns on the tiles, which can be seen in figure 4.3

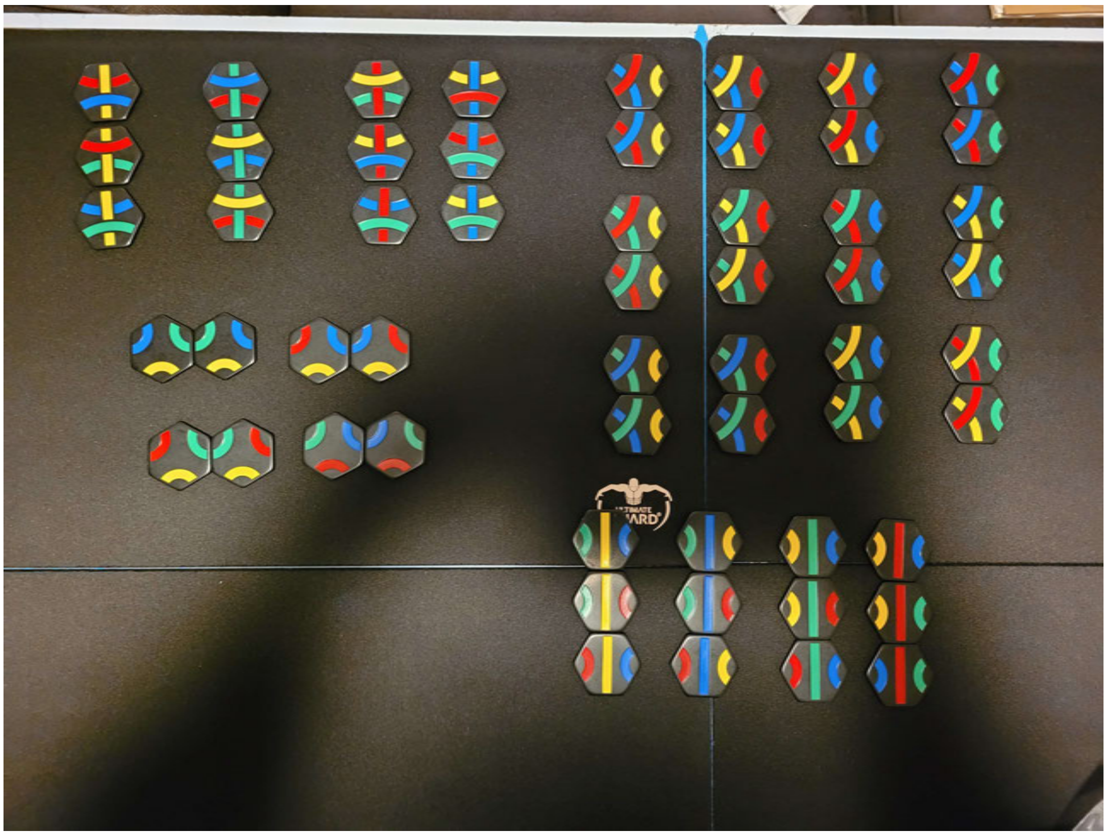


Figure 4.2: All *Tantrix* tiles

with their respective names that are commonly used: *bridge*, *double intersection*, *roundabout* and *single intersection*[3].
It is noteworthy that there is never more than one straight on a tile. In an early version of the *Tantrix* game, a tile with three straights existed, but this tile was removed from the game in 1993[4] and therefore it is not included in this project, either.
Lines can either be red, blue, green or yellow. The different lines on a tile show three of these colors. A specific pattern of a tile always exist in all color combinations. An example of this can be seen in figure 4.4.
In this project, tiles always have the flat-top orientation (see 2.4). For referencing, the edges of a tile are labeled 0 through 5, starting with 0 for the edge at the top of the tile and continuing clockwise.
The pattern and color combination of a tile is encoded in its name. Figure 4.5 shows the digitalized version of all tiles with their respective names. Each character in the name describes the color of the line that crosses a certain edge (B=blue, G=green, R=red, Y=yellow). The first character stands for the color at edge 0, the second character for edge 1, and so forth.

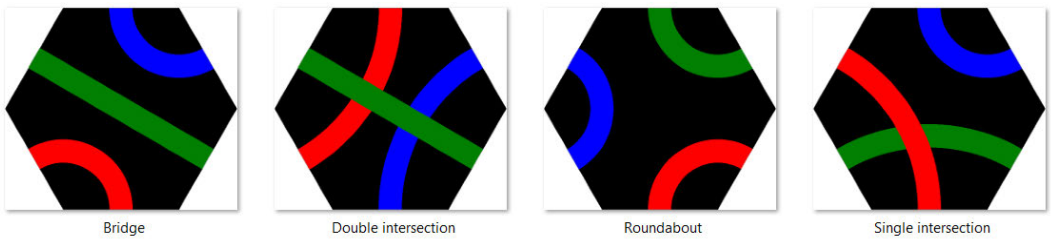


Figure 4.3: *Tantrix* tiles with the four different patterns and their names.

4.6 Constraint File

In order for the algorithm to know what tiles can be placed next to each other, it has to be provided with a file that contains the adjacency rules for the tiles. The structure of this file, as it is used in this project, is based on the constraint files from the parent framework and adapted where needed.
First, the *offsets* are declared, which describe the vectors in a cube coordinate system, that point to all six neighbors of an arbitrary cell on the grid.

[3]FAQ section of the *Tantrix* website. http://www.tantrix.com/english/TantrixUseful.html. Accessed 05/21/24
[4]See footnote 3.

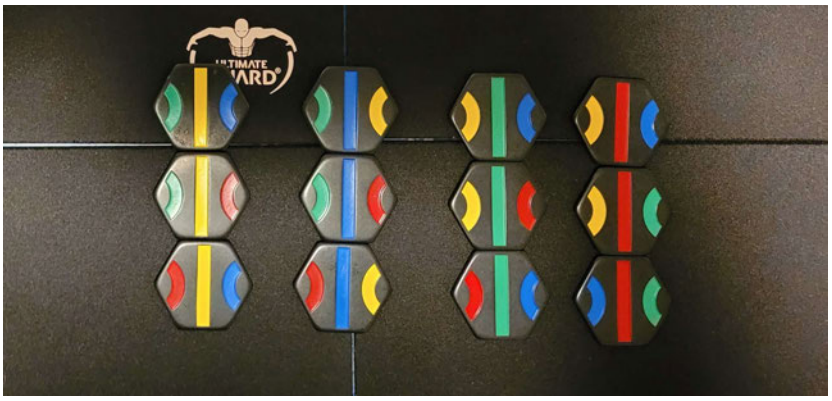


Figure 4.4: *Tantrix* tiles with same pattern.

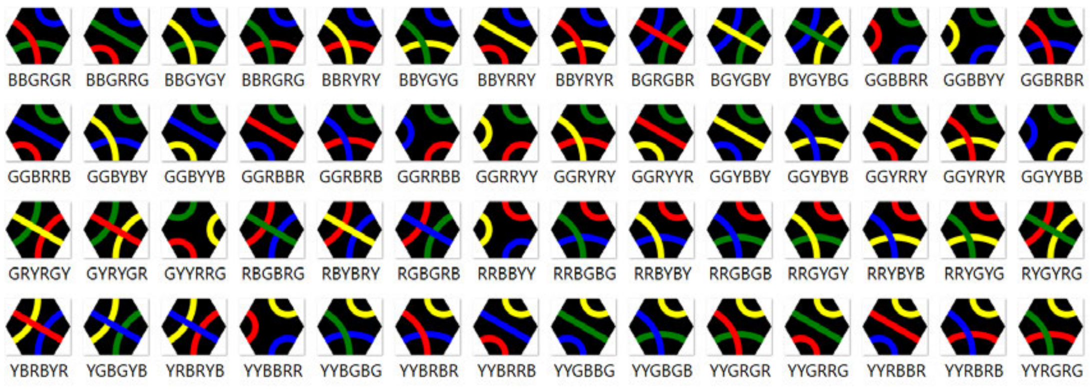


Figure 4.5: Digital *Tantrix* tiles with naming scheme.

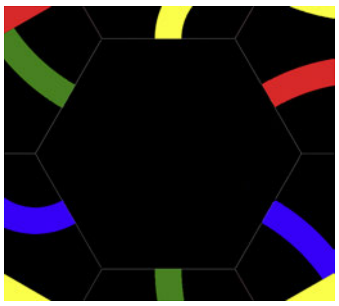
Next, there is an entry for each tile (by name) which contains the filename of that tile, its symmetry type, the weight of the tile, and for each direction a list of possible tiles, that could be placed in that direction. Further details of the constraint file and its creation are discussed in 5.2.

4.7 Normal Mode

Fundamentally, the *normal mode* should just work like the *Simple Tiled Mode*, explained in 2.5. This functionality has already been provided in the framework. However, as mentioned earlier, the implementation in the framework is only able to work with squared tiles. Therefore, there were some changes needed to be made to the code base to enable the use of hexagonal tiles. The whole process of the algorithm can be structured into three main steps:

1. Process the constraint file and create the digital tiles.
2. Run the WFC.
3. Create the output file.

The constraint file only describes possible neighbors for a tile that is in its normal, unaltered orientation. However, the WFC can also rotate a tile before placement. Therefore, the adjacency rules also have to be formulated for each possible orientation that a tile can be in. This happens in step 1. For this purpose, in the framework, tiles were originally rotated in 90° steps. To support hexagonal tiles, the rotation of tiles in 60° steps was implemented. In addition, the possible directions from a tile to its neighbors were expanded from four to six directions.
The WFC itself works mostly as implemented in the framework: First, a grid of cells is built, where each cell could potentially be filled with any of the initial tiles. In contrast to the framework, where the grid has a rectangular shape, this project builds a grid using cube coordinates. Because the dimensions in each direction of the grid are the same, this results in a hexagonal grid.
The algorithm then chooses a cell with the minimal entropy value (of all available cells, this one has the least possible states to choose from) and collapses it to one state (i.e. tile). If at least one cell has no more possible states available, the algorithm restarts, since there is an unsolvable contradiction. Once a cell has been collapsed to one state, this change is propagated through the grid. This procedure is repeated until all cells have only one state left.
While the WFC could mostly be used without modification, special care had to be given when collapsing a cell. In the real world, each tile could only be used once. Applying this


(a) 4 different colors entering a cell.

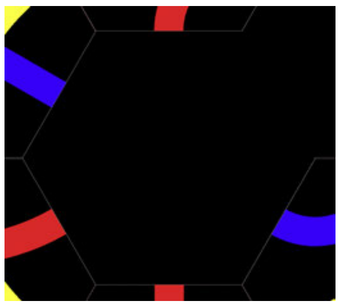

(b) Red enters the same cell three times.

Figure 4.6: Impossible placement scenarios.

restriction also to this project has been considered. However, since there are only 56 unique tiles and the patterns on the tiles themselves are already very restrictive, the decision was made not to include this additional restriction. Another important circumstance, caused by the aforementioned restrictiveness of the tiles, had to be considered during the collapse of a cell: Placing a tile on the grid, even while following the adjacency constraints to the already collapsed neighbors, can make an empty neighboring cell impossible to be solved. This can happen in two situations, which are depicted in figure 4.6. In 4.6a, lines of four different colors point to the same empty cell. In 4.6b, three lines of the same color enter the same cell. As can be seen in figure 4.5, there exists no valid tile that could solve either of the two situations. Therefore, the appearance of those situations had to be prevented. While collapsing a cell, a check is performed to see if the potential tile would create this problem for the empty neighboring cells. If so, this tile is dismissed and a new tile is selected.
For the creation of the output file in step 3, the framework needed to be modified to work with a cube coordinate system, rather than an euclidean coordinate system, again. Furthermore, since an image file is always of rectangular shape, it cannot be completely filled with a hexagon and has empty space in the corners. However, in the creation of the tile images it was ensured that those corners are transparent, which enables the overlapping of tile images in such a way, that hexagons can be placed directly adjacent to each other. Therefore, during the creation of the output file, the tiles had to be placed with some offsets.

4.8 Template Mode

The *template mode* is built on top of the *normal mode*, handling the processing of a template file before the WFC is used. Conceptually, it can be separated into five steps:

1. Create the template file containing the empty grid.
2. Load and process the template file, after the user added a hand drawn shape.
3. Start the normal WFC.
4. Restart the WFC or the whole algorithm, if contradictions arise.
5. Create the output file.

At step 1, the template file is created, which only contains the gray outlines of the tiles. The interiors of the tiles remain white. Together, the tiles form an image of a hexagonal grid with symmetrical dimensions specified by the user.
The user then draws a shape into the template file, using any external image editor. The line to draw the shape has to be pure red (RGB = (255, 0, 0)) and it has to start and end on the same tile. However, the start and end points do not need to align 100%. The template file can then be loaded into the program via the GUI to start the actual processing of the template in step 2. In this step, the algorithm checks every cell of the grid in the template file for the red line drawn in by the user. If such a line is found, it needs to determine what kind of line on a regular *Tantrix* tile best approximates the hand drawn line. For this, two different methods were considered. The first idea was to use an error function whose value should be minimized. This function would consider the differences between the hand drawn line on the tile and each type of line present on the *Tantrix* tiles (straight, curve, corner). However, this method had several drawbacks: To avoid further complicating the calculation of the error function, the hand drawn line would have needed to have a width of just one pixel. Also, for every type of line (straight, curve, corner), a representative tile would be needed whose line, again, is only one pixel wide. Furthermore, determining the path of the hand drawn line on each tile would have required additional image processing.
The alternative approach is to check for each tile that contains a hand drawn line, which two edges of the tile are crossed by that line. The position of those two edges automatically dictates the type of line needed to connect them (e.g. edges 0 and 2 are connected by a curve). This method was ultimately chosen due to its simplicity. Also, it is speculated that both approaches would settle on the same type of line, anyway.
Once the type of line, that is needed, is determined, one tile that contains such a line in yellow is chosen at random from the pool of all tiles. Even though the hand drawn lines are done in red, the tracing in the software is done in yellow. In both situations, those colors were selected for better visibility.
All tiles that are chosen this way are placed, in sequence, on a fresh grid with the same

dimensions as the grid from the template file. The location of the tiles on the new grid is the same as the location of the tiles containing the hand drawn lines in the template file, respectively.
However, before a chosen tile can actually be placed, the same checks that were described in section 4.7 need to be done to prevent unsolvable situations as the ones shown in figure 4.6. Additionally, the placement of a corner tile can introduce another problem which is shown in figure 4.7: During the processing of the template file, the WFC algorithm is not started, yet. This means that the placement of a tile does not reduce the options of the tiles that could be placed in neighboring cells. In most situations this is not problematic, because during the tracing/recreation of the template file, most tiles that are placed next to each other, only connect on the edges that are crossed by the yellow line. However, in a situation like in figure 4.7, where a yellow corner has been placed before, the two tiles that connect to that corner also connect to each other. In figure 4.7, the tile with the thin red outline was initially chosen to be placed next. At this point, an additional check has to be performed to avoid its placement. If a chosen tile is deemed to be invalid, it is no longer considered for that location and a new fitting tile is chosen at random.
The *Tantrix* tiles have one property that has a significant impact on the whole *template mode*: Each line on a tile, no matter the color, always connects two edges on that tile. This means, that each line that appears somewhere on the grid, has to be continued until it either reaches the outer edge of the grid, or until it connects with a line of the same color, forming a closed shape. A line can never simply stop somewhere within the grid.
As stated in the assumptions (4.6), the hand drawn shape provided by the user has to be closed and thereby forming a loop. As a consequence, each colored line in the output image, that enters the shape by crossing the yellow line (which recreates the hand drawn shape), needs to leave the shape at some other location, again. This means, that the total number of lines, that cross the yellow shape in the end, need to be even. If the total number were odd, then there would be one colored line that could not leave the shape anymore and the problem would be unsolvable for the WFC. Looking at the tiles in figure 4.5, it can be seen, that the different types of lines are always crossed by a certain number of other lines: A straight is either crossed by two (differently) colored lines, or not at all. A curve is always crossed by exactly one other line. And a corner is never crossed at all. Consequently, the shape that is drawn by the user must not have an odd number of curves, because then the total number of lines entering/leaving the shape would always be odd, no matter how many straights and curves are otherwise part of the shape.
Furthermore, while recreating the hand drawn shape and selecting fitting tiles to place on the

grid, the color distribution of the lines entering the shape has to be kept in mind. For example, if the last tile to be placed was a yellow curve with a blue line crossing it, but there were only two additional blue lines crossing the shape at other locations, the problem would become unsolvable, again.
To reduce the likelihood of these situations arising, two solutions were considered. For the first one, the number of curves in the hand drawn shape was counted beforehand by the algorithm. Based on that count, a color distribution was generated, ensuring that every color appeared an even amount of times, if at all. Then, when a yellow curve needed to be placed next, the color of the line crossing that curve was already dictated by the distribution. However, predetermining the next color to be used introduced unsolvable situations. For example, if the tile, outlined in red in figure 4.7, was actually required by the distribution to have the red line, the problem could never be solved. A similar situation can be seen in figure 4.8, where a third curve that had a predetermined red line could never be placed, because it would invalidate the neighboring empty cell.
Therefore, another, simpler approach was chosen. Once a tile is placed that has one or two colored lines crossing the yellow line, those colors are noted. Then, if there is an uneven amount of lines of the same color in the shape, the weight of all tiles that have this color crossing the yellow line, is increased. This incentivizes the algorithm to place a tile with that color again, in the future. If a color is present an even number of times, the weights are reset to their normal value. Because this method does not guarantee the creation of a shape that is solvable in the end, the *template mode* uses a restart mechanism. If the WFC reaches a contradiction, it is restarted with the same generated shape. However, if a certain threshold for the number of restarts is reached, the whole template mode is restarted, generating a new version of the recreated shape.

4.9 Architecture

The implementation of the software follows the model view controller architecture (Gharbi et al. [2018]). The general structure of this architecture can be seen in figure 4.9. This architecture allows for a clear separation of different software components, which in turn upholds the maintainabilty of the code. Also, the components can be implemented and tested on their own. Furthermore, the software is easily extendable. In the following sections, the different components of the architecture are explained in more detail.

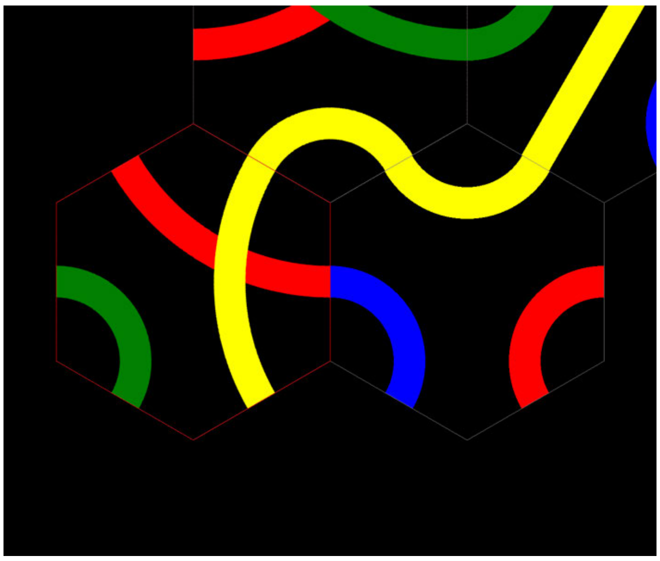


Figure 4.7: Problem when placing a tile next to a corner tile.

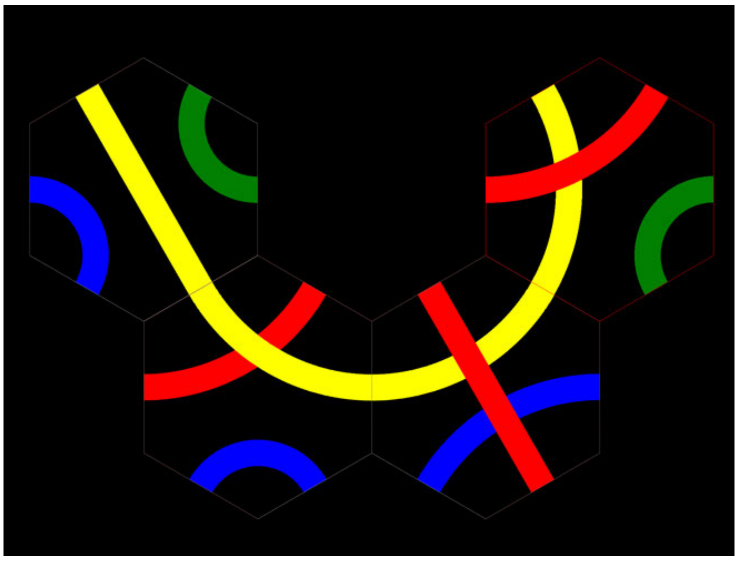


Figure 4.8: Problem with fixed color distribution.

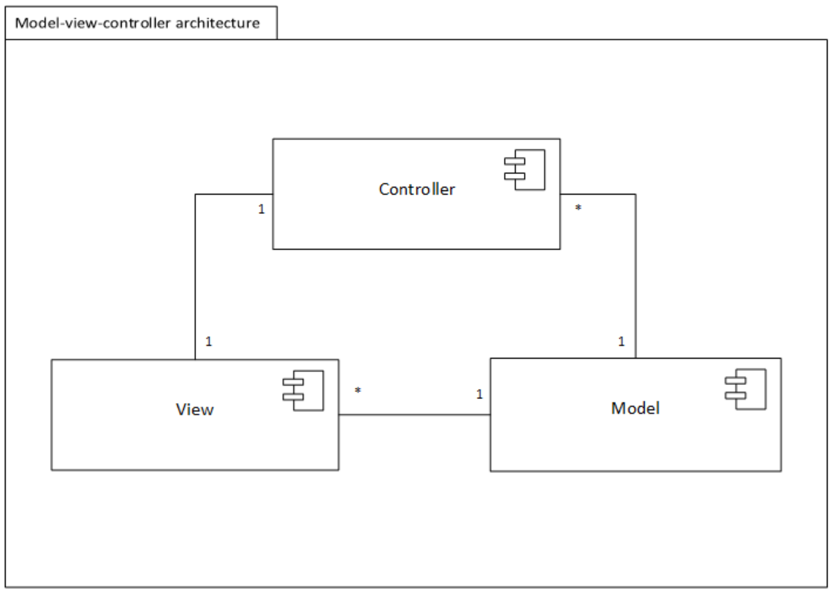


Figure 4.9: Model-view-controller architecture. Image inspired by Gharbi et al. [2018].

Model

The model contains the main logic of the Wave Function Collapse algorithm to run the *normal mode* and additional logic for processing the template file in order to execute the *template mode*.
Furthermore, the model includes data structures that are needed to run the algorithm, like a class representing the cells on the grid and a class for the individual tiles.
In addition, there are several support classes and support methods, for example a class that handles the creation of an output file.

View

The view component provides the GUI for the software that acts as the link between the user and the program. It can be seen in 4.10. Here, the user can input the desired size that the output grid shall have in the end. They can also start the *normal mode* for the WFC via a button.
For the *template mode*, the user can request the creation of an empty template file with the currently selected number of tiles per direction for the grid. Once the template file has been drawn in by the user, they can upload it to the program via another button. Lastly, they can start the template mode via the *Start template mode* button.
In the end, the result of either mode is displayed in a new window.

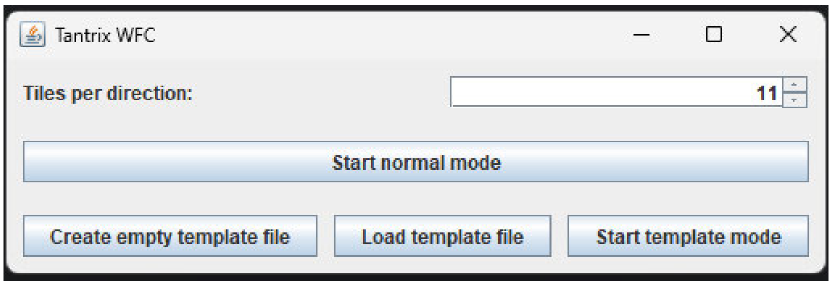


Figure 4.10: GUI for the *Tantrix* WFC.

Controller

The main purpose of the controller is the management of the view and the model components. The controller initializes those components and starts the GUI in a separate thread. It is then able to receive user input from the GUI, process this input, and then initiate required actions in the model.

5 Implementation

This chapter explains how the concepts described in chapter 4 were implemented, using the Java programming language, unless stated otherwise. The level of detail of those explanations depends on whether a class/method was already present in the framework and simply used as is, or if it was modified or added specifically for this project.

5.1 Creation of Digital Tiles

For the creation of the digital *Tantrix* tiles, a Python script, called *tile_creator.py*, was written that takes a text file with the names of all the tiles and creates a PNG file for every different tile.
In order to facilitate the drawing of the tiles, the Pillow library[1] for Python was used, because it provides many modules with methods to create and manipulate images. Additionally, the Numpy library[2] was utilized for easy vector handling.
Besides the main method, the script also contains helper functions whose call sequence is shown in figure 5.1.
The following subsections describe these methods in more detail, but concentrate on highlighting important parts.

5.1.1 Method: main()

After the initialization of some variables such as *hex_size*, *image_width* and *image_height*, the text file, containing the names of all the tiles, is read in and saved as a list of strings called *tile_list*. This list is then iterated over to create an image for each individual tile.
For every tile string, at first a *draw_object* is created that represents a completely transparent image. The transparency is important, because even though the tiles themselves are hexagons, they have to be drawn on a rectangular image. Later, when tiles are placed directly next to each other, corners of the images might overlap other tiles. The transparency prevents the

[1]Website for the Pillow library. https://pillow.readthedocs.io/en/stable/index.html. Accessed 05/24/24
[2]Website for the Numpy library. https://numpy.org/. Accessed 05/24/24

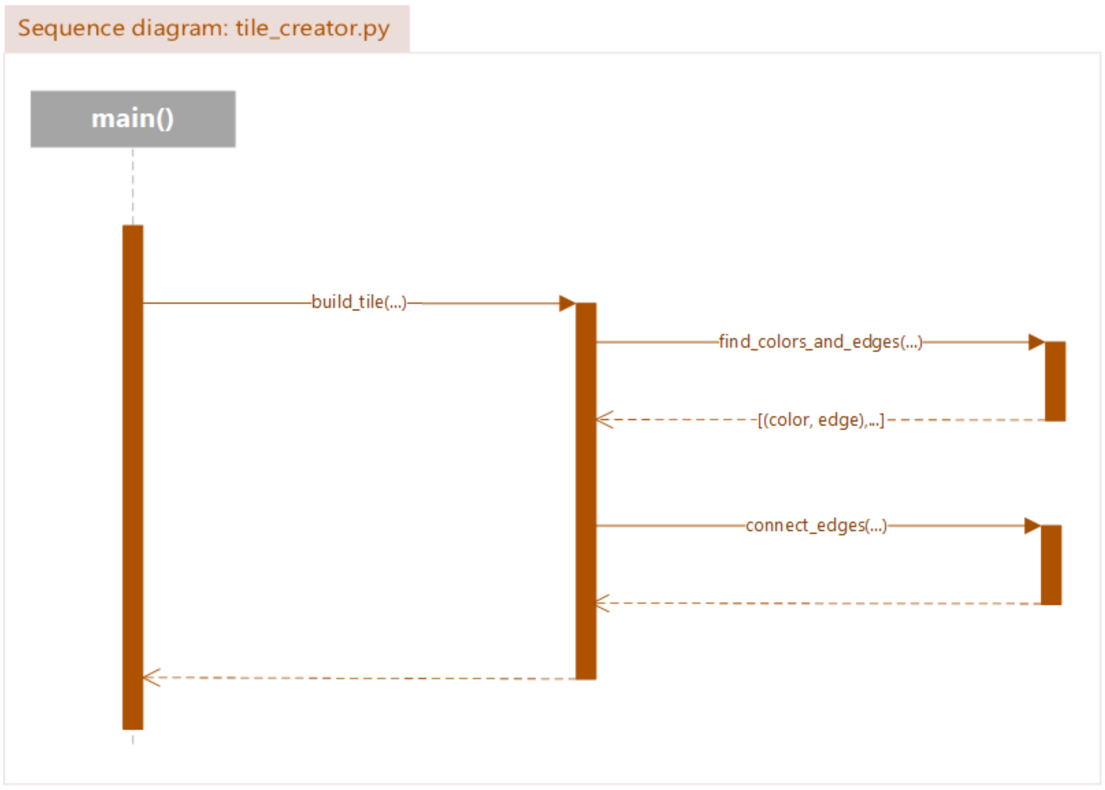


Figure 5.1: Sequence diagram for the tile_creator.py script.

tiles from being overdrawn in those moments.
Next, a black regular polygon is added to the draw_object via a built-in method from the Pillow library. Then, the *build_tile* method is called, with the draw_object as one of the arguments along with the tile_string. This method handles the addition of the correct colored lines to the draw_object, which are dictated by the tile_string. Finally, the image is saved and the next tile_string can be processed.

5.1.2 Method: build_tile(tile_string, draw_object, hex_size, image_width, image_height, middle_point)

First, in this method a dictionary *colors* is defined which maps characters that represent colors to the full names of those colors. For example 'Y' is mapped to "yellow". This is necessary, because the methods that are used to draw the colored lines later do not accept the shorthand version.
Afterwards, the *find_colors_and_edges* method is called with the tile_string as the only argument to figure out what edges need to be connected with which color. Those information are returned to the build_tile method which then delegates them, in sequence, to the *connect_edges* method, where the colored lines are then drawn onto the hexagon.

5.1.3 Method: find_colors_and_edges(tile_string)

This method iterates over the tile_string which it received as an argument. For each character that is encountered and which has not been seen this far in this string, the two indexes are determined, where that character appears in this string. Those two indexes directly encode the two edges that need to be connected by the color represented by the current character. The two indexes are merged into a single string which is then put into a tuple with the current character (e.g. ('R', "03")).
This way, because a tile always contains exactly three lines, three tuples will be found in the end. They are bundled in a list *colors_and_edges* and returned to the calling method.

5.1.4 Method: connect_edges(draw_object, hex_size, image_width, image_height, middle_point, edges_2_connect, color)

In this method, the two edges that are encoded in the argument *edges_2_connect* are connected by a line whose color is defined by the *color* argument. However, before the line can be drawn, some trigonometrical calculations have to be done, which are based on figure 5.2.

First, the line h between the $center$ of the hexagon (given by the argument *middle_point*) and point H has to be determined. This can be accomplished by calculating

$$h = cos(30) * size$$

If the edges have to be connected by a curve or a corner, this is done by using the *arc* method from the Pillow library. This method expects a radius for the arc as one of its parameters. For a curve, this radius can be calculated with

$$radius\_curve = cos(30) * 2 * h$$

For a corner, the formula is

$$radius\_corner = size/2$$

Furthermore, the *arc* method needs a center point, around which the arc will be drawn. For a corner, this is simply the corner point of the hexagon which lies between the two edges that are to be connected.
For a curve, it is the center of the hexagon, moved by an x and y offset (represented by the green lines b and c in figure 5.2). Depending on which edges need to be connected by the curve, the offsets need to be added or substracted from the center point. The offsets can be calculated with

$$b = sin(30) * 2 * h$$

and

$$c = cos(30) * 2 * h$$

If the edges need to be connected by a diagonal straight line across the hexagon, there are different offsets that need to be applied to the center point. This is shown in figure 5.2 by the blue lines a and d, which can be calculated by

$$a = sin(30) * h$$

and

$$d = cos(30) * h$$

Those two points can then be passed to the *line* method from Pillow. Because for each possible pair of edges that need to be connected by a line the potential addition of offsets is unique, the *edges_2_connect* argument is checked in a switch statement, where the matching case will then call either the *line* or *arc* method with the appropriately calculated arguments.

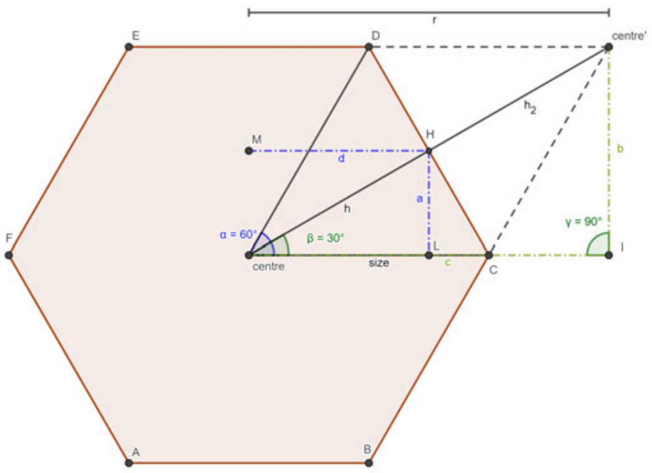


Figure 5.2: Hexagon with construction lines.

5.2 Creation of the Constraint File

The constraint file is implemented as a JSON file [3]. As described in 4.6, the first field has the key *offsets* and as its value a JSON object containing the vectors of the different directions. In this object, each direction serves as a key (e.g., "0"), and the corresponding value is a vector represented as a JSON array (e.g., [0, -1, 1]).
The second field has the key *adjacencyRules* and its value is a JSON object that has an entry for each of the 56 tiles. Every entry consists of another key-value pair, where the tile name is the key and the value is a JSON object describing, in the form of further key-value pairs, the name of the file which contains the image of the tile, the symmetry type of the tile, the weight of the tile, and for each direction a JSON array with the names of the tiles, that can be adjacent in that direction.
Similar to the creation of the digital tiles, a Python script was written in order to construct the

[3]JSON file format. https://www.oracle.com/de/database/what-is-json/. Accessed 05/26/24

constraint file. Again, besides the main method, the script contains some helper functions. All methods are roughly described in the following subsections.

5.2.1 Method: main()

First, a boolean variable *reuse_tiles* is initialized with the default value *True*, which expresses that the algorithm shall be able to place the same tile more than once, later. A value of *False* would reflect the real world scenario, where each tile is only available once. Then the text file, containing the names of all the tiles, is read in and saved as a list of strings called *tile_list*. Furthermore, a dictionary *offsets* is initialized as defined above.
Next, the *tile_list* is iterated over. For each tile, the file name is formed by concatenating the ".png" file extension to the *tile_name*. The symmetry type is set to "L" and the weight to 1. Afterwards, for each direction, the tiles that could be placed adjacent to the current tile have to be determined. For this, the color on the current tile that is facing in the current direction is identified and passed to the *find_tiles_with_color* method. This method returns a list (*matching_tiles*), containing tiles that each have a line of the sought-after color. This list is then iterated over. Each *matching_tile* fits the current tile in exactly two orientations, because the correctly colored line crosses two edges on the *matching_tile*. The method *create_rotated_tile_names* determines if and how the *matching_tile* has to be rotated to fit next to the current tile and returns the names of the *matching_tile* in both orientations (e.g. *RRGGBB60* and *RRGGBB120*), which are then added to the list of valid tiles for the current tile in the current direction. Once every tile is processed this way, the information that was collected is written to a file called *constraints.json*.

5.2.2 Method: find_tiles_with_color(original_tile, edge_color, tile_list, reuse_tiles)

At the beginning of the method, an empty list called *tiles_found* is created, which will hold all tiles that contain a line with the color that is specified in the argument *edge_color*.
To fill this list, the *tile_list* is iterated over, checking for every individual *tile_string* if it contains the color character that is defined in *edge_color*. If it does, it might be a valid tile. However, if *reuse_tiles* is set to *False* and the current tile is equal to the argument *original_tile*, the tile cannot be used. If *reuse_tiles* is *True* or the current tile is not equal to *original_tile*, then there is no problem and the tile can be added to *tiles_found*.
After all tiles have been checked, the *tiles_found* list can be returned to the caller.

5.2.3 Method: create_rotated_tile_names(matching_tile_name, indexes_of_appearances, target_direction, angle_between_edges)

This method gets a tile string *matching_tile_name* as input and a list *indexes_of_appearances* that contains two indexes. Those indexes stand for the two edges of the matching tile which contain the color that has to face in the direction of the tile whose adjacency constraints are currently being formulated. The direction to said tile is passed to the method via *target_direction*. For both indexes, the method now calculates the difference between the index and the *target_direction* and deduces from that difference, how often the matching tile has to be rotated such that the tile is oriented correctly. The number of rotations is then multiplied by *angle_between_edges* and the result concatenated to the *matching_tile_name*.
This way, two versions of the *matching_tile_name* are formed which are returned in a list to the calling method.

5.3 Normal Mode

As mentioned in 4.7, a lot of the code base from the framework could be reused in this project, requiring modifications in some parts. Figure 5.3 gives a rough overview for the classes used in the project. The packages that have a green background were already present in the framework. The package named *tantrixwfc*, with an orange background, was added for the project. The classes *Controller*, *ViewHex* and *HexSupport* were created completely from scratch. The classes *SimpleTiledModel2DHex*, *BufferedImageDrawerHex* and *WaveFunctionCollapseHex* are independent classes, but took inspiration/code from classes from the framework (indicated by the *based on* arrow). However, those classes needed heavy modifications in some parts compared to the original classes. Therefore the decision was made to create completely new classes that are separated from the originals.
The following subsections describe these classes in more detail, highlighting the parts that needed modification or were added for this project.

5.3.1 SimpleTiledModel2DHex

This class handles the processing of the constraint file, initializing the tiles and their constraints. Figure 5.4 shows the corresponding class diagram, highlighting in red text the parts that were modified to support hexagonal tiles. These modifications are explained below.

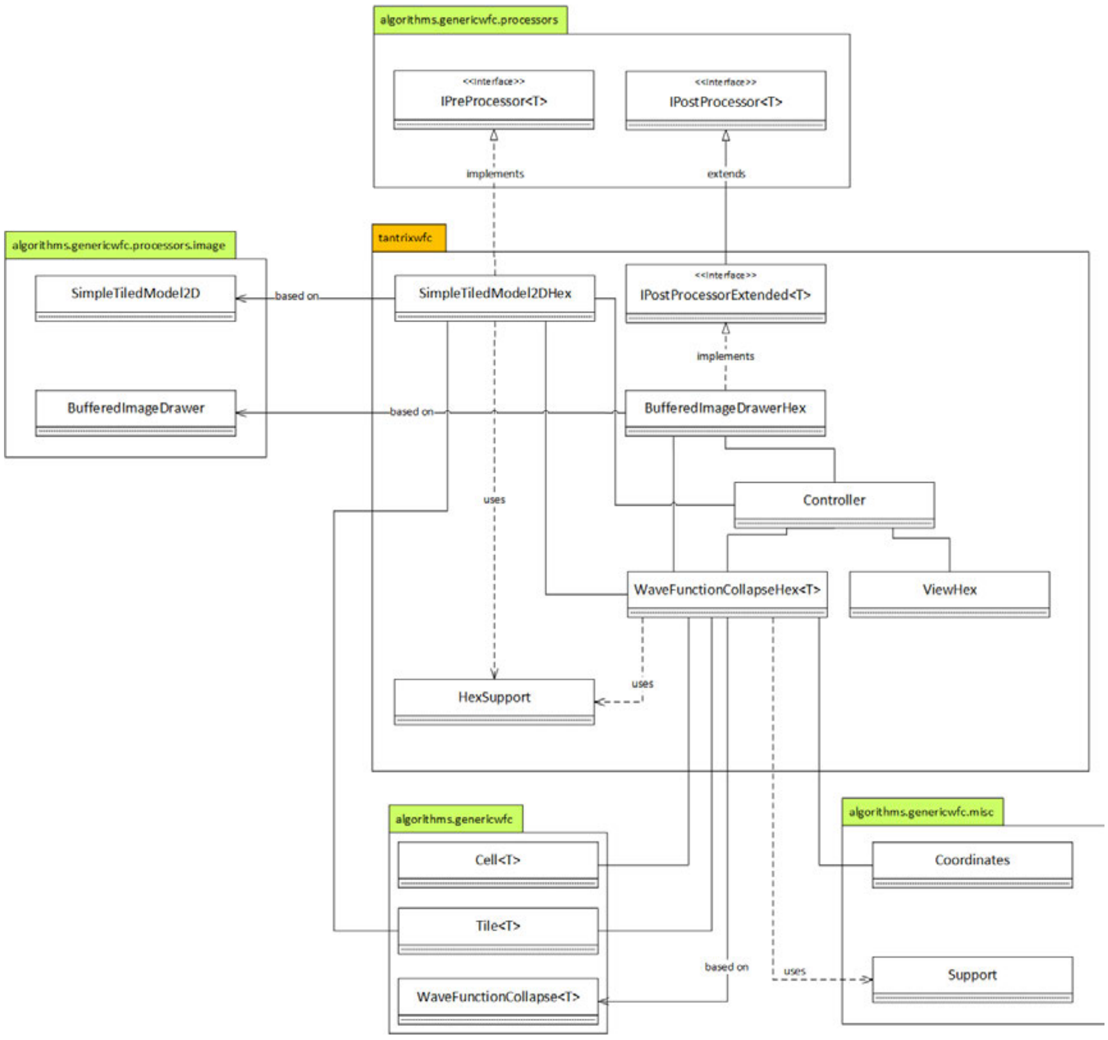
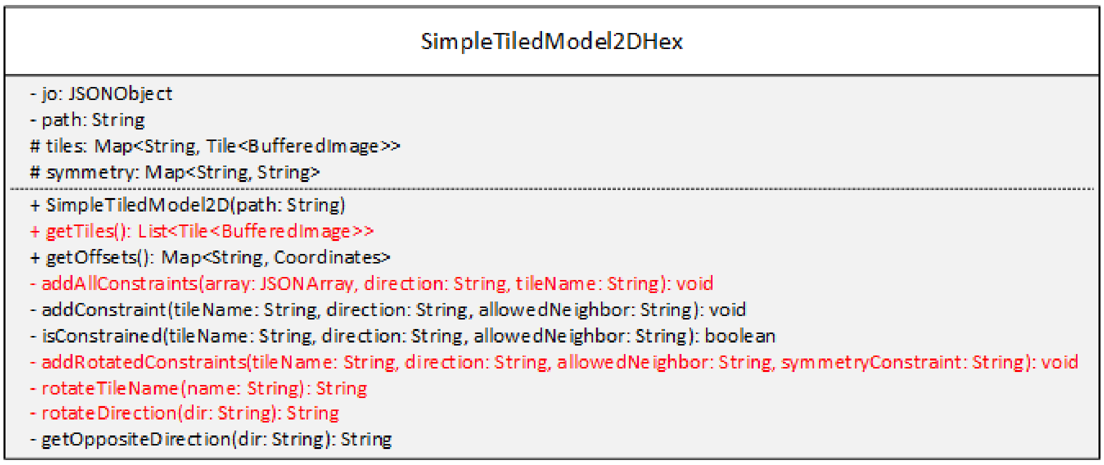


Figure 5.3: Class diagram for the *Tantrix* project. Packages with green background stem from the framework, an orange background marks the package that was added for the project.



```
SimpleTiledModel2DHex
- jo: JSONObject
- path: String
# tiles: Map<String, Tile<BufferedImage>>
# symmetry: Map<String, String>
+ SimpleTiledModel2D(path: String)
+ getTiles(): List<Tile<BufferedImage>>
+ getOffsets(): Map<String, Coordinates>
- addAllConstraints(array: JSONArray, direction: String, tileName: String): void
- addConstraint(tileName: String, direction: String, allowedNeighbor: String): void
- isConstrained(tileName: String, direction: String, allowedNeighbor: String): boolean
- addRotatedConstraints(tileName: String, direction: String, allowedNeighbor: String, symmetryConstraint: String): void
- rotateTileName(name: String): String
- rotateDirection(dir: String): String
- getOppositeDirection(dir: String): String
```

Figure 5.4: Class diagram for *SimpleTiledModel2DHex*. Red text indicates parts that have been modified compared to the original.

Method: getTiles()

In this method, for each tile name in the constraint file, a new tile object is created and added to the *tiles* map. Additionally, the same tile is rotated five times to cover the remaining orientations that the tile could be in. Those versions are added to the *tiles* map as well. In order to rotate the image of a tile, the *rotateImage* method in the *Support* class was overloaded, enabling the rotation by an arbitrary angle instead of fixed 90° steps.
Lastly, for each tile name, the *addAllConstraints* method is called using all six directions, instead of the original four.

Method: addAllConstraints(JSONArray array, String direction, String tileName)

Here, the constraints for a given tile and direction are read in from the constraint file and added to the respective tile object by making a call to the *addConstraint* method. The same adjacency constraints also apply to the rotated versions of the tile. This is handled by calling the *addRotatedConstraints* method.
Furthermore, adjacency constraints are always symmetrical: If tile B can be adjacent in direction 0 to tile A, then tile A can be next to tile B in direction 3. This is also handled in this method, using a newly added *getOppositeDirection* method from the *HexSupport* class and modified versions of the *rotateTileName* and *rotateDirection* methods.

Method: addRotatedConstraints(String tileName, String direction, String allowedNeighbor, String symmetryConstraint)

Constraints that apply for a tile in its original orientation also apply for the rotated versions of the same tile. This is handled in this method by rotating the tile five times, adjusting the *direction* accordingly by calling the *rotateDirection* method and adding the *allowedNeighbor* as a constraint under the adjusted direction to the rotated tile.

Method: rotateTileName(String name)

Rotating the tile name of a tile is done by appending the angle of rotation to the original tile name. This method checks if the tile has already been rotated and adds 60° (instead of the original 90°) to the current rotation (e.g. *RRGGBB60* becomes *RRGGBB120*).

Method: rotateDirection(String dir)

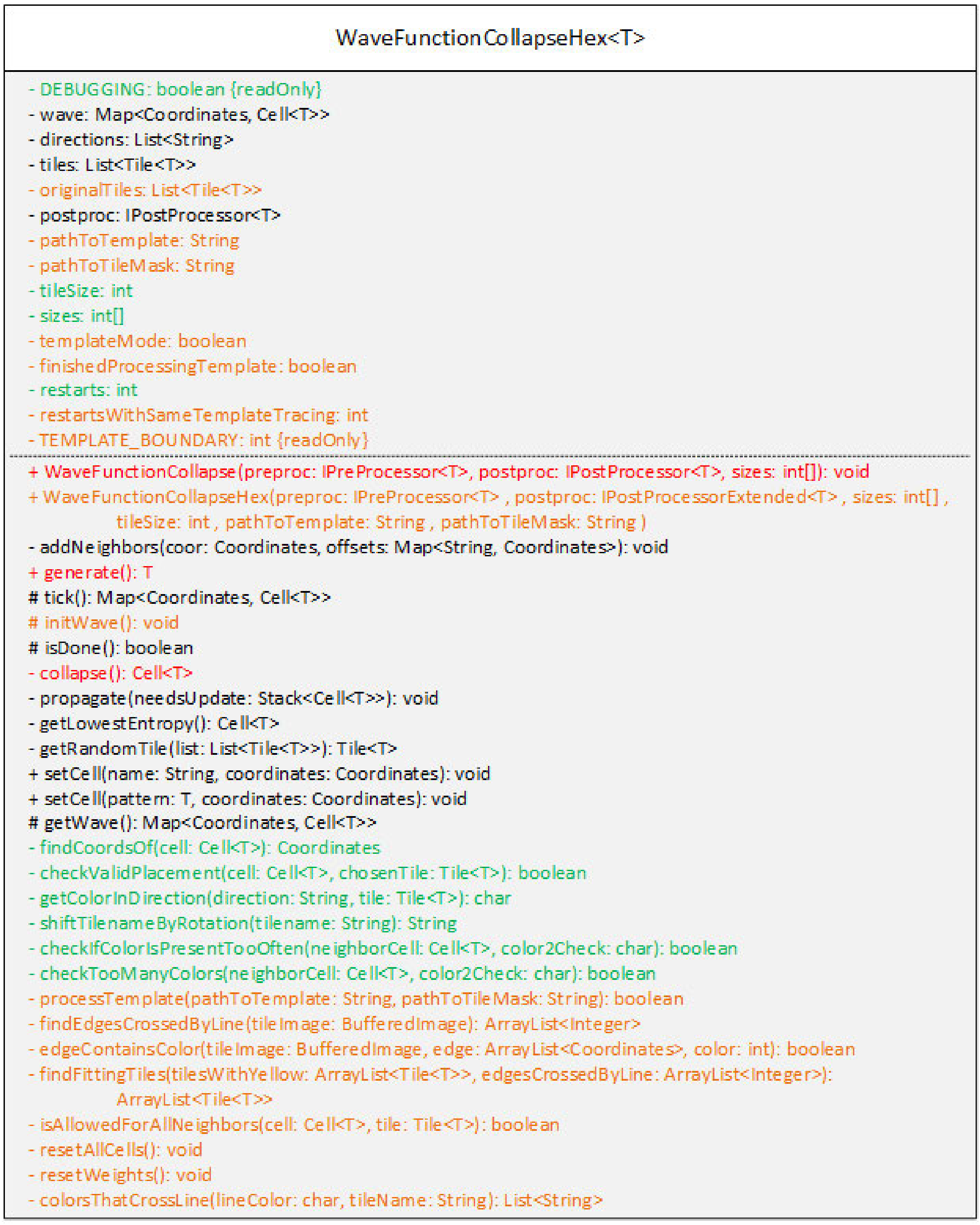
This method simply returns the direction that is next to the input direction in a clockwise manner. In the original version, it mapped, for example, *top* to *right*. Now it maps, for example, *0* to *1*.

5.3.2 WaveFunctionCollapseHex

As the name suggests, the *WaveFunctionCollapseHex* class implements the WFC algorithm. Figure 5.5 shows the associated class diagram. The parts written in orange can be ignored while considering the *normal mode*, because they only concern the *template mode*. The methods written in red and green are observed in more detail in the following, because they are either modified versions from the original or newly added.
For a better understanding of the general flow of the algorithm, figure 5.6 shows the rough sequence of function calls that are triggered during execution of the algorithm. However, this diagram does not include every call to helper functions, that might occur.

Constructor: WaveFunctionCollapse(IPreProcessor<T> preproc, IPostProcessorExtended<T> postproc, int[] sizes)

The main purpose of the constructor is to initialize the *wave* member variable which represents the grid that is to be filled in with tiles, later. Every coordinate on the grid is associated with a cell. At the beginning, each cell could potentially collapse to any of the tiles provided to the algorithm. The coordinates in this version of the algorithm are in cube coordinate format.



```
WaveFunctionCollapseHex<T>

- DEBUGGING: boolean {readOnly}
- wave: Map<Coordinates, Cell<T>>
- directions: List<String>
- tiles: List<Tile<T>>
- originalTiles: List<Tile<T>>
- postproc: IPostProcessor<T>
- pathToTemplate: String
- pathToTileMask: String
- tileSize: int
- sizes: int[]
- templateMode: boolean
- finishedProcessingTemplate: boolean
- restarts: int
- restartsWithSameTemplateTracing: int
- TEMPLATE_BOUNDARY: int {readOnly}

+ WaveFunctionCollapse(preproc: IPreProcessor<T>, postproc: IPostProcessor<T>, sizes: int[]): void
+ WaveFunctionCollapseHex(preproc: IPreProcessor<T> , postproc: IPostProcessorExtended<T> , sizes: int[] ,
        tileSize: int , pathToTemplate: String , pathToTileMask: String )
- addNeighbors(coor: Coordinates, offsets: Map<String, Coordinates>): void
+ generate(): T
# tick(): Map<Coordinates, Cell<T>>
# initWave(): void
# isDone(): boolean
- collapse(): Cell<T>
- propagate(needsUpdate: Stack<Cell<T>>): void
- getLowestEntropy(): Cell<T>
- getRandomTile(list: List<Tile<T>>): Tile<T>
+ setCell(name: String, coordinates: Coordinates): void
+ setCell(pattern: T, coordinates: Coordinates): void
# getWave(): Map<Coordinates, Cell<T>>
- findCoordsOf(cell: Cell<T>): Coordinates
- checkValidPlacement(cell: Cell<T>, chosenTile: Tile<T>): boolean
- getColorInDirection(direction: String, tile: Tile<T>): char
- shiftTilenameByRotation(tilename: String): String
- checkIfColorIsPresentTooOften(neighborCell: Cell<T>, color2Check: char): boolean
- checkTooManyColors(neighborCell: Cell<T>, color2Check: char): boolean
- processTemplate(pathToTemplate: String, pathToTileMask: String): boolean
- findEdgesCrossedByLine(tileImage: BufferedImage): ArrayList<Integer>
- edgeContainsColor(tileImage: BufferedImage, edge: ArrayList<Coordinates>, color: int): boolean
- findFittingTiles(tilesWithYellow: ArrayList<Tile<T>>, edgesCrossedByLine: ArrayList<Integer>):
        ArrayList<Tile<T>>
- isAllowedForAllNeighbors(cell: Cell<T>, tile: Tile<T>): boolean
- resetAllCells(): void
- resetWeights(): void
- colorsThatCrossLine(lineColor: char, tileName: String): List<String>
```

Figure 5.5: Class diagram for the *WaveFunctionCollapseHex* class. Methods and fields written in red are modified versions from the original *WaveFunctionCollapse* class. A green text color marks newly added parts. Black means unmodified. Orange are parts that are either modified or added but are only important for the *template mode*.

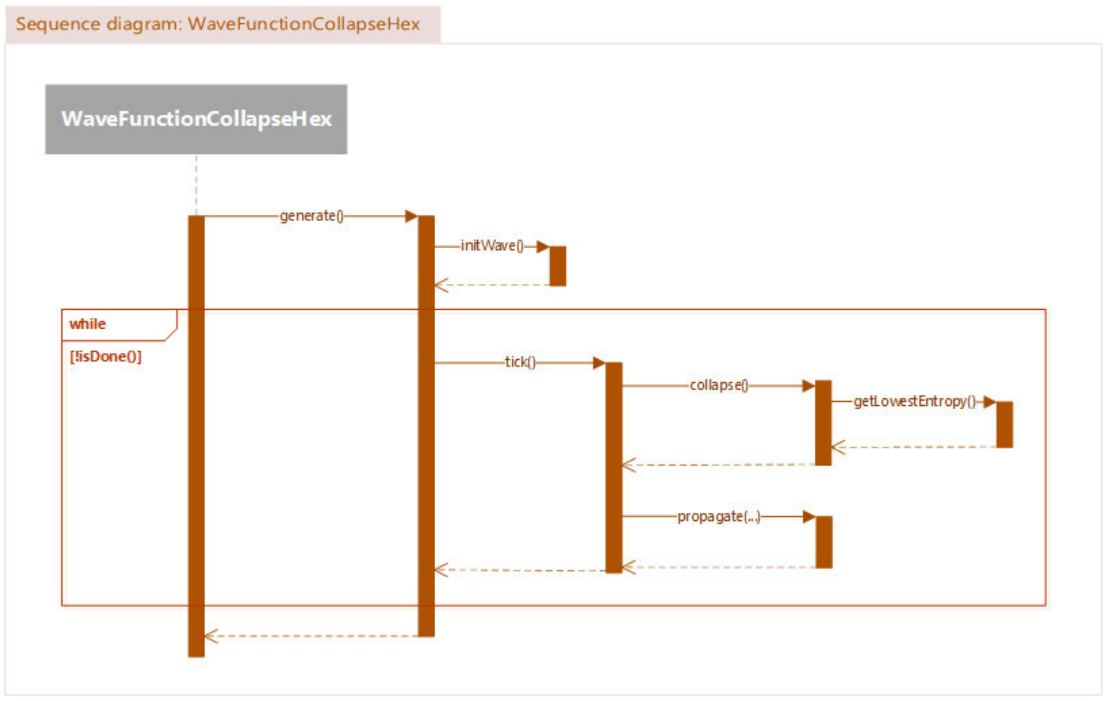


Figure 5.6: Sequence diagram which shows the fundamental method calls during execution of the WFC. Note: This diagram does not include every call to helper functions.

While initializing the *wave*, which is done by iterating through the coordinate space defined by the dimensions in each direction of the grid, it is important that the coordinate values for each direction add up to zero. Otherwise, the coordinate is not a valid cube coordinate and does not lie on the grid. Therefore is has to be skipped.

Method: generate()

This method is the entry point to start the execution of the algorithm. It calls the *initWave* method in order to propagate any predefined tiles for certain cells through the grid, if such initial constraints were given to the algorithm.
Afterwards, the *tick* method is called in a loop, until each cell is left with only one state, meaning that there is one distinct tile assigned to each cell.
In this version of the *generate* method, once the algorithm is done, the name of the output file is extended with the amount of restarts that the algorithm needed to finish its execution.

Method: collapse()

The *collapse* method starts with a call to the *getLowestEntropy* function to find the cell that has the lowest entropy and therefore the smallest amount of possible states that it could collapse to. If the return value from *getLowestEntropy* is *null*, at least one cell has no more possible states available to collapse to. This indicates a contradiction, prompting a restart of the algorithm.
Otherwise, if a lowest entropy cell is found, one of the possible tiles is chosen at random. In the original version of this method, the chosen tile was directly placed (by associating the cell with said tile). However, as mentioned in 4.7, while working with the *Tantrix* tiles, the chosen tile cannot simply be placed but the placement has to be checked for validity in advance. Therefore, the *checkValidPlacement* method is called. If it returns *True*, the tile can be used. Otherwise, the tile is removed from consideration for this cell for this moment and another possible tile is chosen at random. If no suitable tile is found this way, the algorithm is restarted.

Method: checkValidPlacement(Cell<T> cell, Tile<T> chosenTile)

For the given *cell*, the neighboring cells in each direction are considered. If a neighbor is neither *null* (which indicates the edge of the grid) nor already collapsed to one final state, the color of the line on the *chosenTile* that faces in the direction of the neighbor is determined by calling the *getColorInDirection* method. This color is passed, together with the neighboring cell, to the *checkIfColorIsPresentTooOften* method to check if the same color already points at least twice to that neighboring cell from other tiles. If the method returns *False*, a second method, *checkTooManyColors*, is called with the same arguments to see if the color would be the fourth distinct color that is pointing towards the neighboring cell. Only if that method also returns *False* is the placement considered valid.

Method: getColorInDirection(String direction, Tile<T> tile)

This helper function determines the color of the line leaving a given tile in a specified direction. Theoretically, this color is already encoded in the tile name string and could be determined by looking at the character that is at the same index as indicated by *direction* (e.g. the color for direction *1* on the tile *RRGGBB* is *R*).
However, the tile could also be rotated. A rotation of a tile is not encoded in the sequence of the characters in the tile name, but the angle of rotation is added to the end of the tile name. Therefore, the *shiftTilenameByRotation* method is called in advance to get the tile name that actually represents the current rotation of the tile (e.g. *RRGGBB60* would become *BRRGGB*). Then the method described above can be applied to determine the color of the line.

Method: shiftTilenameByRotation(String tilename)

This helper function shifts the name of a tile according to the rotation. For this purpose, the angle of rotation is first extracted from the *tilename*, removing it from the name in the process. Then the angle of rotation is divided by 60 in order to calculate how often the tile name needs to be shifted to get a string that correctly represents the orientation of the tile.
During shifting, a character at the end of the string gets moved to the front, therefore wrapping around. Once the correct number of shifts are performed, the new name can be returned.

Method: checkIfColorIsPresentTooOften(Cell<T> neighborCell, char color2Check)

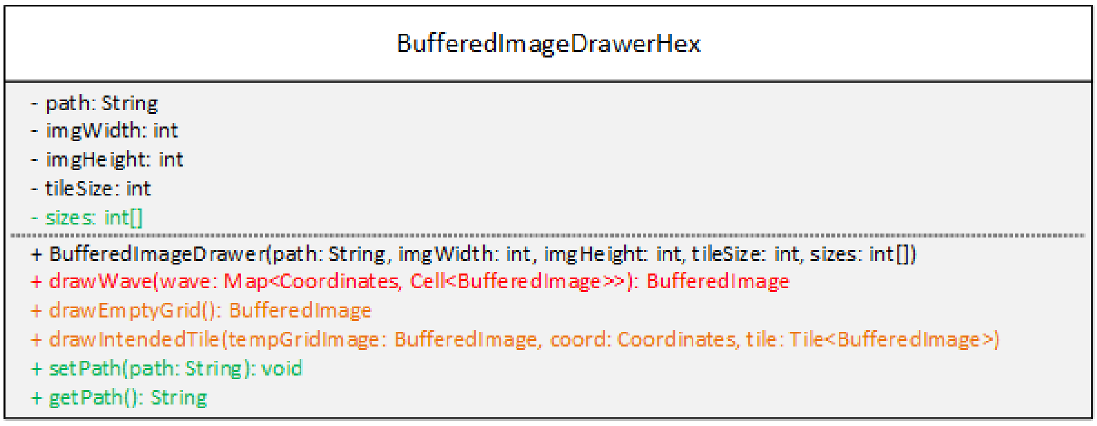
The *neighborCell* that is passed to the method has not been collapsed yet. It has to be determined, if the color described by *color2Check* is already pointing towards this cell from any of its neighbors at least twice. Therefore, the neighboring cells in each direction are considered. If a neighbor exists and the cell is already collapsed to one tile, the color on that tile that is facing the original *neighborCell* is determined. If it equals *color2Check*, a counter *colorPresent* is incremented by one.
Once *colorPresent* exceeds a value of two, the method can stop and return *True*, because the color would already be present too often. If all directions get checked and the counter stays below two, *False* can be returned.

Method: checkTooManyColors(Cell<T> neighborCell, char color2Check)

Like in the *checkIfColorIsPresentTooOften* method, the neighborhood of the *neighborCell* is checked. However, this method records, which colors are facing this cell.
In the beginning, *color2Check* is added to an empty list called *colorsPresent*. While iterating the neighbors of the *neighborCell*, if a new color is encountered that points in the direction of the *neighborCell*, it is added to *colorsPresent*.
If the size of *colorsPresent* ever exceeds three, the method can return *True*, because then too many different colors would be facing towards the *neighborCell*. If all directions get checked and the size of *colorsPresent* stays below four, *False* can be returned.

5.3.3 BufferedImageDrawerHex

The main purpose of this class is to draw the completed grid and save it to a PNG file. The class diagram is depicted in 5.7. Even though the constructor is marked as *modified*, this modification only encompasses the addition of a new parameter *sizes*. Otherwise, for the *normal mode*, only the modified *drawWave* method is important.



```
BufferedImageDrawerHex

- path: String
- imgWidth: int
- imgHeight: int
- tileSize: int
- sizes: int[]

+ BufferedImageDrawer(path: String, imgWidth: int, imgHeight: int, tileSize: int, sizes: int[])
+ drawWave(wave: Map<Coordinates, Cell<BufferedImage>>): BufferedImage
+ drawEmptyGrid(): BufferedImage
+ drawIntendedTile(tempGridImage: BufferedImage, coord: Coordinates, tile: Tile<BufferedImage>)
+ setPath(path: String): void
+ getPath(): String
```

Figure 5.7: Class diagram for the *BufferedImageDrawerHex* class. Methods and fields written in red are modified versions from the original *BufferedImageDrawer* class. A green text color marks newly added parts. Black means unmodified. Orange are parts, that are either modified or added but are only important for the *template mode*.

Method: drawWave(Map<Coordinates, Cell<BufferedImage» wave)

This method iterates through all valid cube coordinates that belong to the grid. For every coordinate, it gets the image belonging to the final state (tile) of the cell located on that coordinate. This image of the tile is drawn onto a bigger image *result* at the position defined by the coordinate. However, because the images that show the tiles are actually rectangular and not hexagonal themselves, they have to be placed with an offset. The new position of the upper left corner of the tile image can be calculated with:

$$x = (int)((centerOfGridX + (tileSize * (3/2 * q))) - tileSize)$$

and

$$y = (int)((centerOfGridY + (tileSize * (\sqrt{(3)}/2 * q + \sqrt{(3)} * r))) - ((\sqrt{(3)} * tileSize)/2))$$

The values for *centerOfGridX* and *centerOfGridY* are the halfway points of the width and height of the output image. The variable *tileSize* is passed into the constructor and is defined as *size* in 2.6. The variables q and r stand for the current position in the respective dimension of the coordinate.
Once every cell of the grid has been processed and drawn, the *result* image can be saved as a PNG file to the location specified in the *path* member variable.

5.4 Template Mode

This mode extends the *normal mode* with the functionality of providing the template file (containing an empty grid) to the user and processing the drawn in template file afterwards. The *SimpleTiledModel2DHex* class did not need any more changes than described in 5.3. However, the *WaveFunctionCollapseHex* and *BufferedImageDrawerHex* classes needed additional modifications, which are outlined in the following.

5.4.1 WaveFunctionCollapseHex

On top of running the normal WFC as described in 5.3, this class also provides the functions to process the template file. Again, figure 5.5 shows the member variables and methods that were either modified or added to this class. For the *template mode*, only the parts written in orange need to be considered.

Constructor: WaveFunctionCollapseHex(IPreProcessor<T> preproc, IPostProcessorExtended<T> postproc, int[] sizes, int tileSize, String pathToTemplate, String pathToTileMask)

This is an overloaded version of the constructor that was used in the *normal mode*. The first addition to this version is the inclusion of three more parameters: *tileSize*, *pathToTemplate* and *pathToTileMask*.
The constructor starts with a call to the constructor used in the *normal mode* to do the normal preparations. Then, the *wave* is prepared further by calling the *initWave* method. Afterwards, the arguments provided are used to initialized the respective member variables. The flag *templateMode* is set to *True*, in order to indicate that the *template mode* is currently executed. Finally, the *processTemplate* method is called in a loop, until the template file is successfully processed.

Method: initWave()

This method resets all cells in the wave to their initial states. First, it is checked if the *templateMode* flag is set to *True*, the template file has been successfully processed at least once (*finishedProcessingTemplate == True*) and the variable *restartsWithSameTemplateTracing* is greater than *TEMPLATE_BOUNDARY*. In this case, the WFC was not able to calculate a valid solution for the currently generated tracing of the template and a full restart needs to be initiated. This includes a call to the *resetAllCells* method, setting *finishedProcessingTemplate* to *False*, resetting *restartsWithSameTemplateTracing* to zero and calling the *processTemplate* method again, until it returns successfully.
Afterwards, the states for all cells that are marked as *resettable*, are reset to include the whole current tile set saved in *tiles*. Note that at this point *tiles* does not include any tiles that have yellow lines on them anymore.
Lastly, the *wave* is searched for any cells that have their *resettable* flag set to *False*. Those are cells that already have a predetermined state (for example after processing the template file). For each of those cells, the final state is then propagated through the grid in order to adjust the adjacency constraints of other cells.

Method: resetAllCells()

This helper function resets all cells to the original state, even the ones that are normally not *resettable*. This has to be done when the processing of the template file fails or when the number of restarts using the same generated tracing exceeds the threshold defined by

TEMPLATE_BOUNDARY.
Afterwards, the states include all the initial tiles again, even the yellow tiles that were removed from *tiles* in the *processTemplate* method.

Method: processTemplate(String pathToTemplate, String pathToTileMask)

This method starts with reading in the images for the *templateFile* and *tileMask* from the locations defined by both arguments, respectively.
Next, all tiles that contain a yellow line are filtered from *tiles* and saved in a new ArrayList called *tilesWithYellow*. Those tiles are further categorized in a map called *yellowTilesSorted*. The key set of this map contains *R*, *G*, *B*, *GR*, *BR*, *BG* and *noCrossing*. Every key describes the color(s) of the line(s) that cross the yellow line on the tiles stored under that key. For each tile in *tilesWithYellow*, the *colorsThatCrossLine* method is called to determine where the tile has to be placed in the map.
Next, the method searches for all tiles in the template file that contain a hand drawn line. For this, it iterates through all the coordinates that belong to the grid. For each coordinate, a subimage is taken from the *templateFile* at that location and stored in a temporary BufferedImage called *tileImage*. The subimage has the same size as the images of the tiles provided to the algorithm and therefore depicts the tile present in the *templateFile* at the current coordinate. However, because the subimage is of rectangular shape, it also includes parts of neighboring tiles in the corners of the image. For further processing of the *tileImage*, those parts need to be blacked out, first. This is done by comparing the *tileImage* with the *tileMask* (seen in figure 5.8) pixel by pixel. Every pixel in the *tileMask* that is black also gets set to black in the *tileImage*.
Afterwards, the *findEdgesCrossedByLine* method is called for the current *tileImage* to find the two edges on the tile, that are connected by the hand drawn line. Those edges are returned from the *findEdgesCrossedByLine* method and stored in an ArrayList *edgesCrossedByLine*. If this list is empty, the tile does not include any hand drawn line and the next coordinate can be assessed. However, if *edgesCrossedByLine* does contain edges, a fitting tile needs to be found next, that could be placed on the current coordinate in order to recreate the hand drawn line. Such a tile features a yellow line that connects the same edges. Therefore, the *findFittingTiles* method is called, which searches all *tilesWithYellow* for appropriate tiles and returns them as an ArrayList, which is stored in *fittingTiles*.
Next, one tile is selected at random from *fittingTiles*. As described in 5.3 for the *collapse* method, the placement for the tile has to be checked first by calling *checkValidPlacement*. Additionally, a second check has to occur that is described in 4.8 and is implemented in the *isAllowedForAllNeighbors* method. If the tile cannot be placed, it is removed from *fittingTiles*

and a new tile is chosen randomly. If no tile can be found this way, the *processTemplate* method needs to be restarted.
Once a valid tile is found, it is set in the *wave* using the *setCell* method. This automatically collapses the cell to one state, containing the fitting tile, and marks the cell, which belongs to the current coordinate, as not *resettable*.
There is a high chance that the tile, which was just placed, has colored lines that cross the yellow line. As described in 4.8, this circumstance has to be considered to ensure the solvability of the generated shape. Therefore, the *colorsThatCrossLine* method is called, whose return value are all the colors that cross the yellow line on this tile. Those colors are saved in a list *colorsThatCrossShape*. This list is then iterated over and for each color present a respective boolean variable (*needToUseBlue*, *needToUseRed* or *needToUseBlue*) is flipped. If that variable shows *True* after flipping, the weight of every tile that has a line in the respective color crossing the yellow line is set to 100. This increases the chance of those tiles being chosen in the *getRandomTile* method. This helps in creating a template tracing which has an even amount of lines for every color crossing the yellow shape. If the variable shows *False* after flipping, the weights of the tiles for that color are reset to the normal value of 1.
Finally, after processing each coordinate on the template file, all *tilesWithYellow* are removed from *tiles*, because for visual clarity no more yellow tiles should be used in the normal WFC anymore. Furthermore, the weights of all tiles are reset to the normal value (1) by calling *resetWeights*. Afterwards, the *processTemplate* method can return *True*, indicating to the caller that the template was successfully processed.

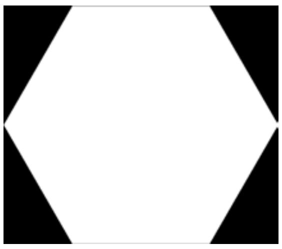


Figure 5.8: Tile mask with black corners.

Method: findEdgesCrossedByLine(BufferedImage tileImage)

This method finds the edges of a tile that are crossed by the red line drawn across the tile (if present) and returns the indexes of those edges in a list. The indexes start at 0 for the top edge of the hex and are increased clockwise.
As a first step, the coordinates of every corner of the hexagon in the *tileImage* are calculated by

calling the helper function *findCoordsOfAllCorners*, which is implemented in the *HexSupport* class. Then, for every pair of adjacent corners, the pixels that belong to the edge, which connects those two corners, are calculated and stored in an ArrayList named like the respective edge (e.g. *edge0*). These calculations are done using the Bresenham algorithm (Janser and Luther [1992]), which is implemented in the *calcBresenhamLine* method in the *HexSupport* class.
Next, for each edge the method *edgeContainsColor* is called to check if the red line crosses that edge. If it does, the name of the edge is added to the *edgesCrossedByLine* list, which is returned at the end of the method.

Method: edgeContainsColor(BufferedImage tileImage, ArrayList<Coordinates> edge, int color)

This helper function determines if a given edge in an image of a hexagon contains a given color. This is done by iterating every pixel, defined in the *edge* argument, in the *tileImage* and comparing its color value with the *color* argument. In case of a match, *True* can be returned. Otherwise, *False* is returned.

Method: findFittingTiles(ArrayList<Tile<T» tilesWithYellow, ArrayList<Integer> edgesCrossedByLine)

This method searches the list *tilesWithYellow* for tiles, where the yellow line connects the same edges as defined by the parameter *edgesCrossedByLine*. To achieve this, the name of every tile in *tilesWithYellow* is shifted by calling *shiftTilenameByRotation* to correctly represent the orientation of that tile. Each tile name can then further be examined. If the two characters at the indexes defined in the argument *edgesCrossedByLine* in the tile name are both a 'Y', this tile is a correct fit and can be added to the list *fittingTiles* which is returned in the end.

Method: isAllowedForAllNeighbors(Cell<T> cell, Tile<T> tile)

Before a tile can be placed, in addition to the normal *checkValidPlacement* call, in the *template mode* it has to be checked if the *tile* does correctly connect to all tiles that are already placed in the neighborhood of the *cell*. Therefore, for every tile that has already been placed in the neighborhood, the set of tiles that is allowed in the direction of the *cell* is queried. If the *tile* is contained in this set for every neighbor, the tile can be safely placed.

Method: resetWeights()

This method simply iterates through all tiles in *tiles* and sets their weight to 1.

Method: colorsThatCrossLine(char lineColor, String tileName)

This method checks for a given tile (represented by *tileName*), which other colors cross a specified colored line (*lineColor*) on that tile.
First, the two indexes in the *tileName*, where the character *lineColor* appears, are determined. Then the substring in the *tileName* between those indexes is formed. If another colored line crosses the specified line, the character describing the colored line can only appear once in the substring. If it appears twice, it is a corner and does not cross the specified line. Each character that appeared once is added to a list *colorsThatCrossLine*.
Finally, the characters in this list are ordered alphabetically, before returning the list to the caller.

5.4.2 BufferedImageDrawerHex

As can be seen in figure 5.7, indicated by the orange text font, the two methods *drawEmptyGrid* and *drawIntendedTile* were added to this class specifically for the *template mode*.

Method: drawEmptyGrid()

This method works very similarly to the *drawWave* method. However, while iterating through all possible coordinates on the grid, instead of drawing the appropriate tile from the finished *wave*, this method always draws the same image of an empty tile. This image consists of a white hexagon with a gray outline. The result is an empty grid like the one shown in figure 5.9.

Method: drawIntendedTile(BufferedImage tempGridImage, Coordinates coord, Tile<BufferedImage> tile)

For debugging purposes, it was desirable to have a way of showing and highlighting a tile that is intended to be placed next on the partially filled out grid.
The *tempGridImage* is the image of the unfinished grid, that can be created by calling the *drawWave* method. The argument *tile* is the tile that is supposed to be placed next and *coord* is the location on the grid, where the *tile* should be placed.
Besides calculating the x and y offsets that were already described earlier in the *drawWave*

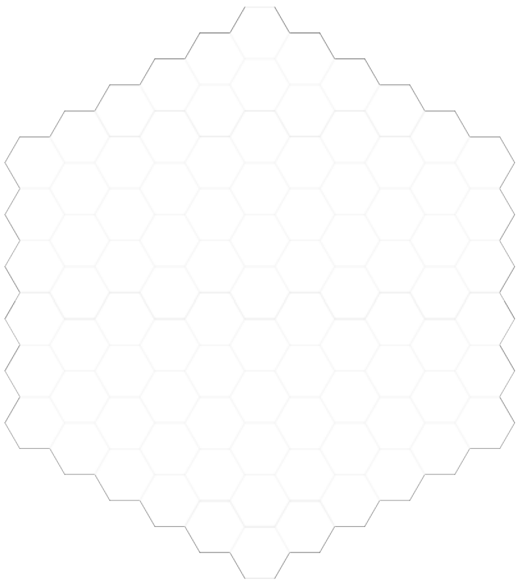


Figure 5.9: An empty grid.

method, this function simply draws the image of the *tile* to the appropriate coordinate in the *tempGridImage* and draws a red outline around the tile in order to highlight it.

5.5 Controller

The *Controller* class forms the link between the user (via the *View*) and the WFC algorithm (in the *normal mode* and the *template mode*). Its structure is shown in figure 5.10.

5.5.1 Constructor: Controller(ViewHex view)

The constructor initializes the member variables that do not already have a default value. Furthermore, it sets the various behaviors for the buttons displayed in the GUI.

5.5.2 Method: main(String[] args)

The main method initializes a new *ViewHex* and a new *Controller* and starts a new thread for the *view*.

5.5.3 Methods: startTemplateMode()/startNormalMode()

The main purpose of both methods is to start the *normal mode* or the *template mode*, respectively. For this, a new *WaveFunctionCollapseHex* object is created by calling the appropriate constructor.

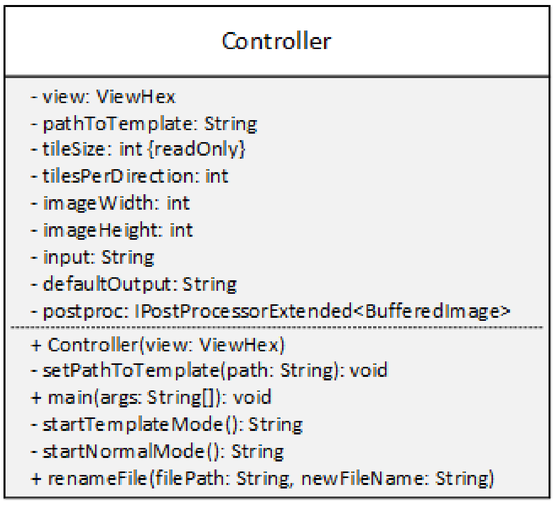

Controller
- view: ViewHex
- pathToTemplate: String
- tileSize: int {readOnly}
- tilesPerDirection: int
- imageWidth: int
- imageHeight: int
- input: String
- defaultOutput: String
- postproc: IPostProcessorExtended<BufferedImage>
+ Controller(view: ViewHex)
- setPathToTemplate(path: String): void
+ main(args: String[]): void
- startTemplateMode(): String
- startNormalMode(): String
+ renameFile(filePath: String, newFileName: String)

Figure 5.10: Class diagram for the *Controller* class.

The algorithm is started by calling the *generate* method.
Otherwise, both methods only handle organizational tasks such as measuring the execution time of the algorithm and adding important information to the name of the output file, like the grid size or a time stamp.

5.5.4 Method: renameFile(String filePath, String newFileName)

Some information, like the grid size, can already be appended to the name of the output file before that string is handed to the algorithm. However, the total execution time can only be added to the file name, after the algorithm is finished. At that time, the output image is already created in the file system. This method provides a means of renaming said file with a new name.

5.6 ViewHex

This class implements standard functionality of a GUI. Therefore, besides providing the class diagram in figure 5.11, it shall not be explained in more detail at this point.

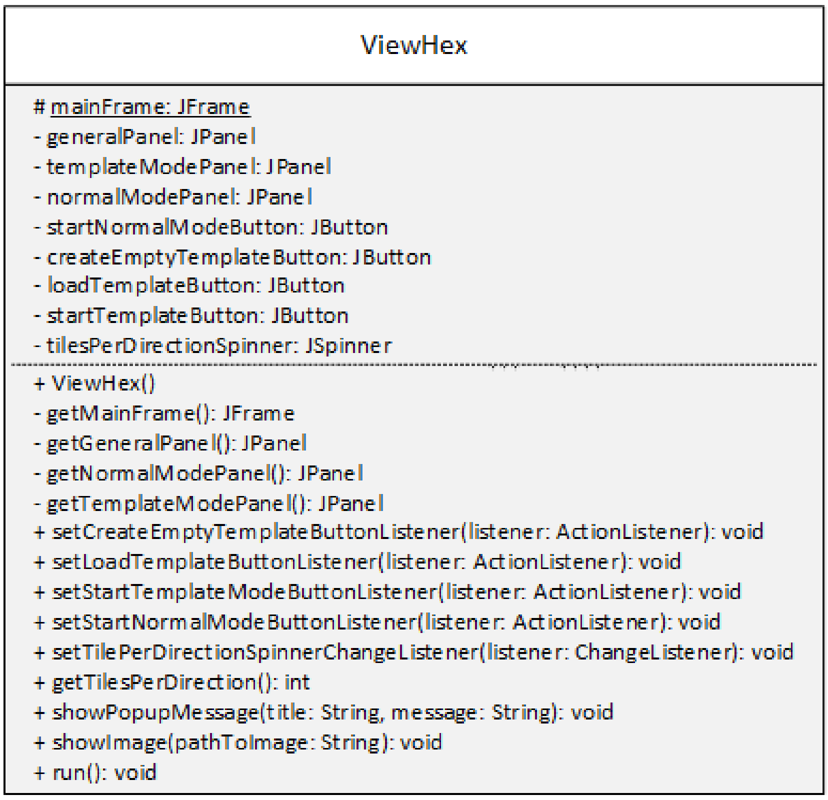

ViewHex
# mainFrame: JFrame
- generalPanel: JPanel
- templateModePanel: JPanel
- normalModePanel: JPanel
- startNormalModeButton: JButton
- createEmptyTemplateButton: JButton
- loadTemplateButton: JButton
- startTemplateButton: JButton
- tilesPerDirectionSpinner: JSpinner
+ ViewHex()
- getMainFrame(): JFrame
- getGeneralPanel(): JPanel
- getNormalModePanel(): JPanel
- getTemplateModePanel(): JPanel
+ setCreateEmptyTemplateButtonListener(listener: ActionListener): void
+ setLoadTemplateButtonListener(listener: ActionListener): void
+ setStartTemplateModeButtonListener(listener: ActionListener): void
+ setStartNormalModeButtonListener(listener: ActionListener): void
+ setTilePerDirectionSpinnerChangeListener(listener: ChangeListener): void
+ getTilesPerDirection(): int
+ showPopupMessage(title: String, message: String): void
+ showImage(pathToImage: String): void
+ run(): void

Figure 5.11: Class diagram for the *ViewHex* class which implements the GUI.

6 Evaluation

In this chapter, the implementation of the concept is evaluated. It is checked, which functional requirements (4.2.1) could and which could not be met. If a requirement was broken, a short reason shall be given. In order to evaluate the performance of the implementation, the *normal mode* and *template mode* were run with different grid sizes (and difficulties for the *template mode*). For each setting, 10 runs were performed. The hardware that was used for these runs is as follows: LENOVO L390 Yoga (20NT0011GE) with Intel(R) Core(TM) i5-8265U CPU @1.60GHz, Intel(R) UHD Graphics 620, 16GB RAM. The program is started using IntelliJ IDEA 2023.2.5 (Ultimate Edition) on Windows 11 Pro.
For both modes, the two timestamps used to measure the duration of a run were taken directly before the constructor of the WFC and directly after the subsequent call of the *generate* method. The correctness of an output was determined by visual inspection of the output image. Implementing a function to validate the output image did not seem feasible in the scope of this project.

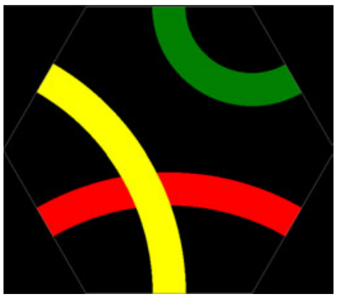
6.1 Normal Mode

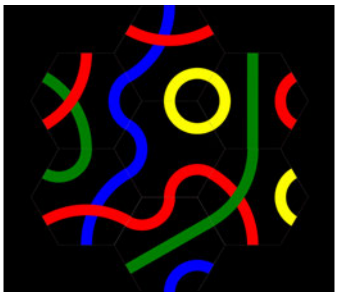
Figure 6.1 shows the correctly generated output images for different grid sizes. It can be seen that requirements 1-8 are fulfilled for the *normal mode*. Requirement 9, concerning the restart of the WFC in case of a contradiction, cannot be checked here, because the WFC was always able to create a correct output image without the need for restarts during this mode.
However, with increasing grid sizes, the *normal mode* faces another problem: Starting at grid sizes of 31x31x31 and above, the program sometimes runs out of heap space. This behavior can be seen in figure 6.2. The heap size (set in IntelliJ) for the system that the tests were performed on is 2GB. Once this memory is exceeded, the algorithm cannot continue and the execution is canceled. Taking a look at figure 6.3, which shows the average sizes of the output files for different grid sizes, it does not come as a surprise that the algorithm reaches its limits sooner or later for growing grid sizes. For example, the average file size for the output of a 41x41x41 grid is 16.39Mb. However, this is only the image that is created at the end of the algorithm. During execution, the WFC works internally with BufferedImages, each cell having multiple

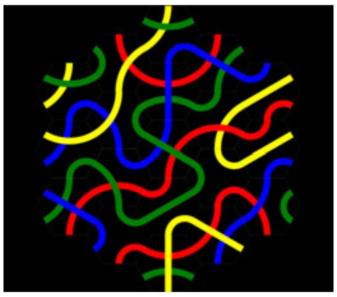
of those BufferedImages assigned to it before it is collapsed to one final state. For a 41x41x41 grid, there are 1261 such cells. As figure 6.2 shows, for a grid size of 43x43x43, the algorithm was only able to finish in two out of ten runs and for a grid size of 51x51x51 it could not finish at all. Concerning this problem, the algorithm could probably be improved by optimizing it further. Also, a different data structure might be advantageous.
Apart from these memory errors, the algorithm performs well during *normal mode*. When considering all output images that could be finalized, the success rate was 100% (i.e. no output image showed incorrectly placed tiles), without the need for any restarts. The average duration for different grid sizes over ten runs can be seen in 6.4. As expected, the execution time rises with increasing grid sizes, but with about 50 seconds for a 41x41x41 grid it stays withing a reasonable window. This satisfies requirement 28.

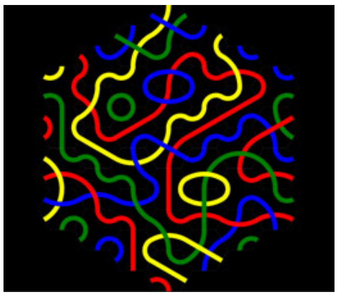
6.2 Template Mode

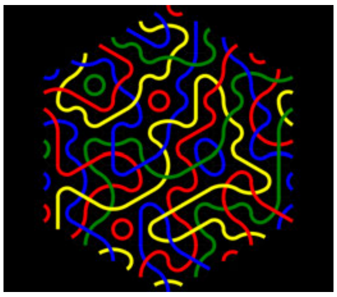
For the *template mode*, as requested in requirement 14, the user can start the creation of an empty template file via a button. Figure 4.10 shows the respective button and figure 6.5 displays two exemplary results of that action. Figure 4.10 also shows the other buttons/fields that are demanded in requirements 12, 13, 15 and 16. The two types of error messages specified in requirements 17 and 18 are shown in figure 6.6 and 6.7. The automatic displaying of the output image, once the algorithm is finished (Req.19), can be seen in figure 6.8.
In order to evaluate the implementation of the *template mode*, different template files with grid sizes between 3x3x3 and 21x21x21 were created. For each grid size, the template file was filled out three times with three kinds of shapes of different difficulties: easy, medium, hard. Those difficulties are roughly defined by how many tight corners and narrow passages are present in the shapes, although they remain largely subjective. Grid sizes above 21x21x21 were not chosen, because drawing the template shape became too imprecise due to the small size of the individual tiles when the image was completely zoomed out. Zooming in was not a good option, because that would have required a lot of panning the image while drawing, making it harder to draw a closed loop. Furthermore, each template file had to contain a shape that is theoretically solvable. Therefore, the number of curves in the shape has to be even, which has to be considered during drawing. This is further complicated when the image needs to be zoomed in and panned.
Figure 6.9 shows these template files and figure 6.10 displays results that correctly recreated the templates. The results show that requirements 1-6 and 8 are also satisfied for the *template mode*. Requirements 10 and 11, which are specific for the *template mode*, are also met.

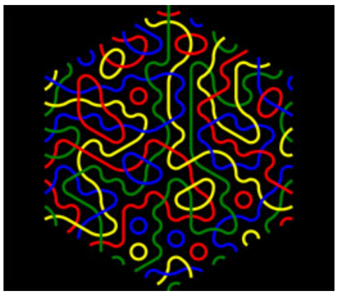

(a) 1x1x1

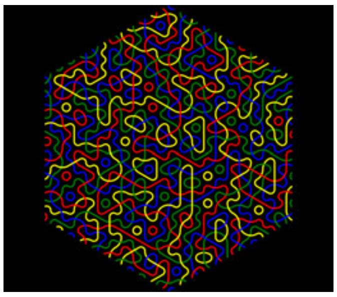

(b) 3x3x3

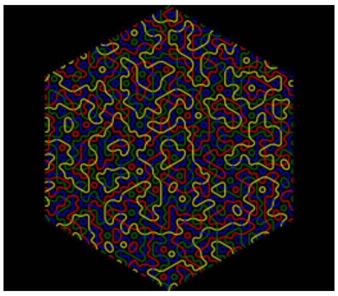

(c) 5x5x5

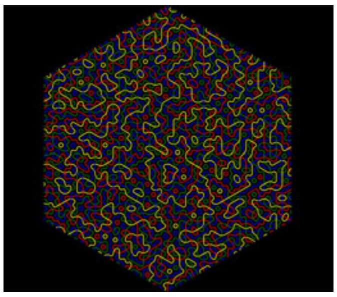

(d) 7x7x7


(e) 9x9x9


(f) 11x11x11


(g) 21x21x21


(h) 31x31x31


(i) 41x41x41

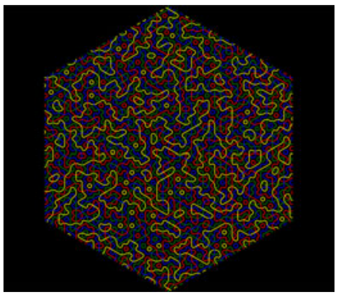

(j) 43x43x43

Figure 6.1: Correct output from the *normal mode* for different grid sizes.

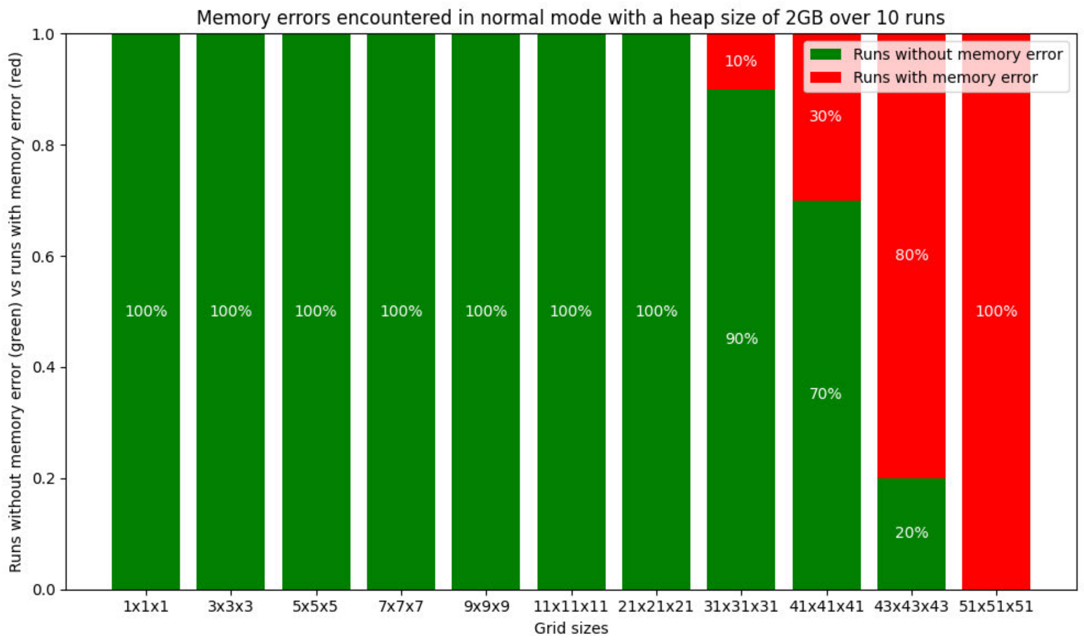


Figure 6.2: Memory errors for different grid sizes in *normal mode*.

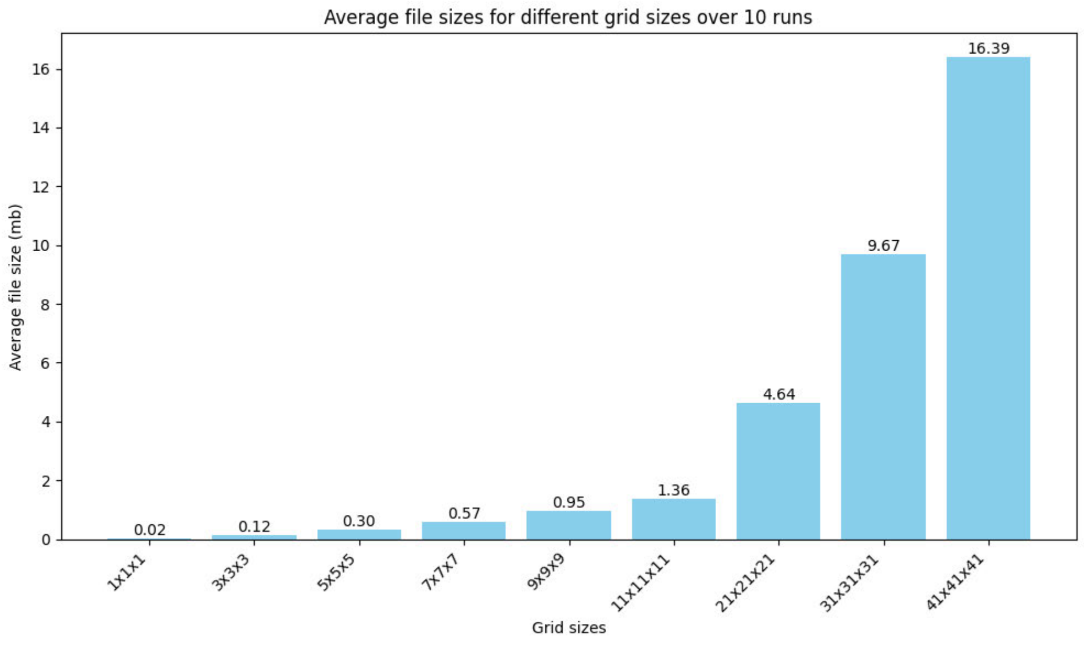


Figure 6.3: Average file sizes for the output files of different grid sizes in *normal mode*.

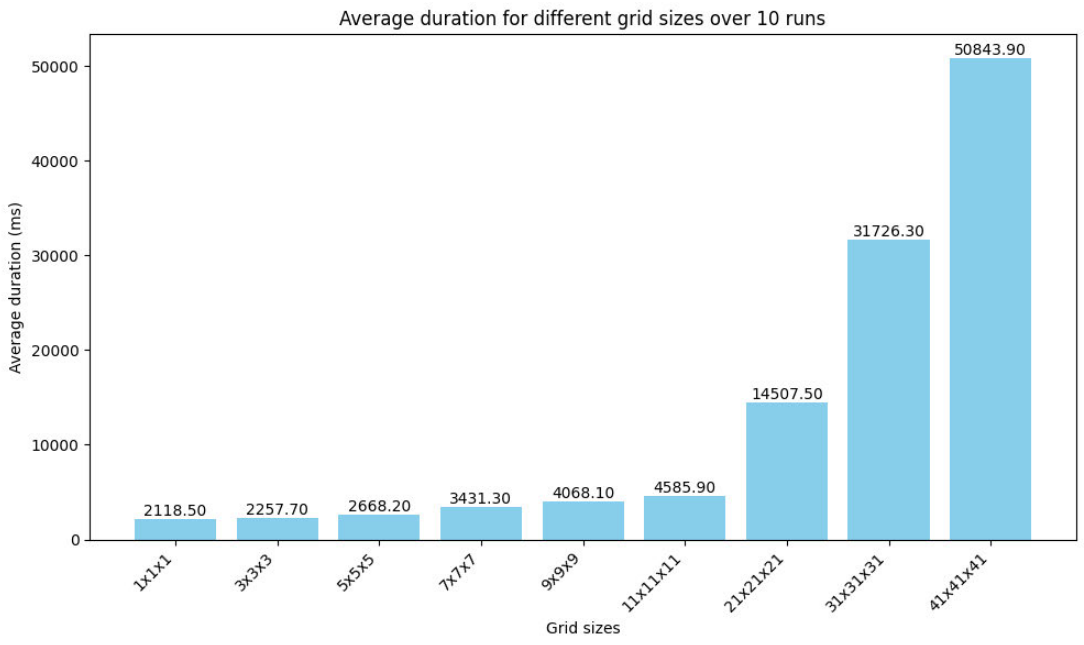


Figure 6.4: Average duration of the WFC for different grid sizes in *normal mode*.

However, requirements 7 and 9, which state that the output shall not have any contradictions and if contradictions are encountered, that the algorithm should restart, are not always fulfilled during *template mode*. This can be seen in figure 6.11. The incorrect tiles in the interior of the shape should not have been placed, but rather a restart should have been initiated. This inability of the algorithm to spot a contradiction is a bug that could not be solved during work on the project. Furthermore, this bug does not always occur: Figure 6.12 shows the average amount of restarts for different grid sizes and difficulties during *template mode*. Clearly, sometimes the algorithm is able to spot contradictions and initiate restarts. During debugging, it was observed that the recreation of the template sometimes caused many cells in the interior of the shape to automatically collapse to one state. This depends on the level of restrictiveness of the shape. For example, narrow passages in the interior of the shape (see left part of the shape in figure 6.11), are sometimes deterministically collapsed to one state as a consequence of the selection of the tiles that trace the template. It is suspected that those (automatically collapsed) tiles might not be correctly propagated through the grid once the main part of the WFC starts. This problem has to be further investigated in the future.
As mentioned in 4.8, the recreated shape needs to have a valid color distribution for the lines that cross the yellow shape for it to be solvable. In order to avoid an organizational overhead,

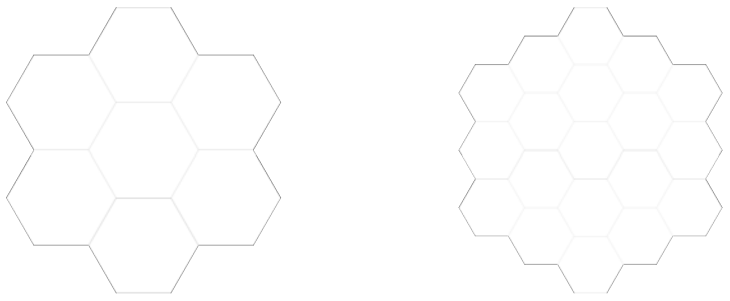
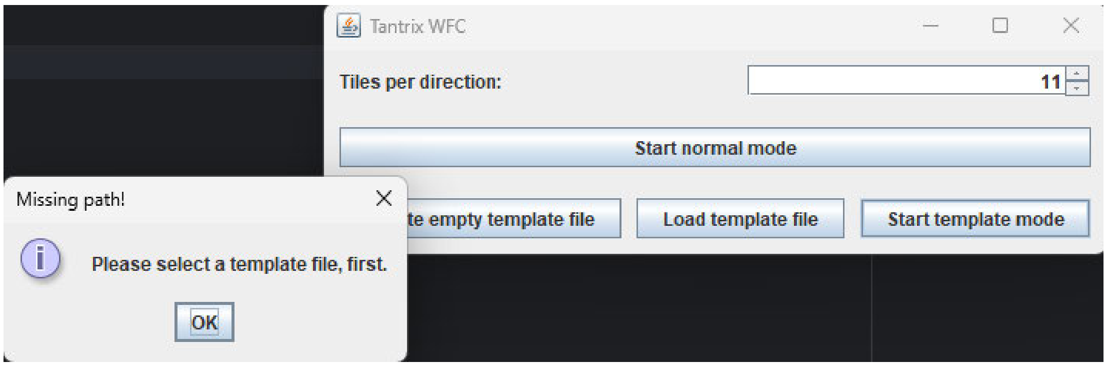

(a) 3x3x3 (b) 5x5x5

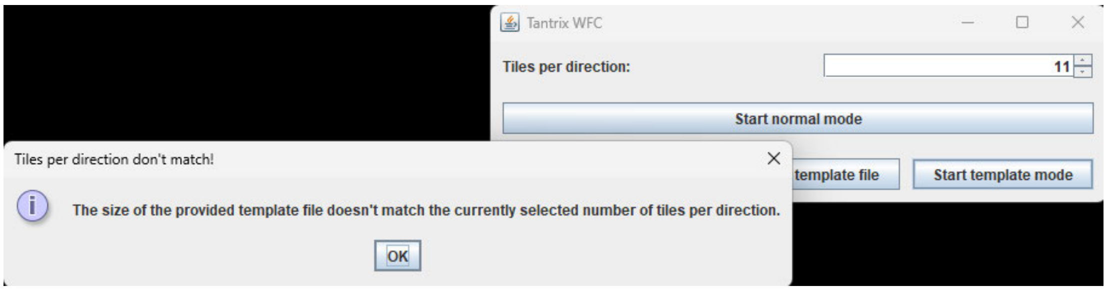
Figure 6.5: Two exemplary template images in *template mode* for different grid sizes.

a simple approach was chosen to attempt to create such a distribution. Figure 6.13 shows the results of the *template mode* for ten runs for each grid size and difficulty setting. The green bar shows how many runs yielded a correct output image, the red bar displays how many outputs were incorrect. The blue bar depicts how often the recreated shape was solvable with regard to the color distribution. Gray stands for unsolvable color distributions. Considering the simplicity of the method to ensure correct color distributions, the results are satisfactory. A success rate of 5/10 or less for some grid sizes might seem bad. However, it should be kept in mind that the algorithm would normally spot contradictions in the output and restart the algorithm, if the bug was not present. After a set amount of normal restarts, a full restart would be initiated which also includes the new recreation of the template shape. Therefore, without the bug, the blue bar should theoretically always cover 100% of the runs. A possible drawback would be, that the execution of the *template mode* could take up significantly more time than the *normal mode*. Therefore, a more effective method to create a valid color distribution would be advantageous.
The results of the time measurements for the current state of the *template mode* can be seen in figure 6.14. As expected, the processing of the template image slows the algorithm down in relation to the *normal mode*. Compared to the worst delay in execution time of around 1.8 seconds mentioned in the related work (3), the implementation of the *template mode* performs worse. The biggest difference between the *normal mode* and the *template mode* is about 8 seconds for a grid size of 21x21x21. However, the execution time is still reasonable and satisfies requirement 28.
Therefore, it can be concluded that the *template mode*, theoretically, fulfills all requirements. Fixing the mentioned bug should enable the algorithm to consistently produce correct output images.


Missing path!

Please select a template file, first.

Figure 6.6: Error message when no template file is selected.


Tiles per direction don't match!

The size of the provided template file doesn't match the currently selected number of tiles per direction.

Figure 6.7: Error message when the grid size in the template file does not match the currently selected grid size.

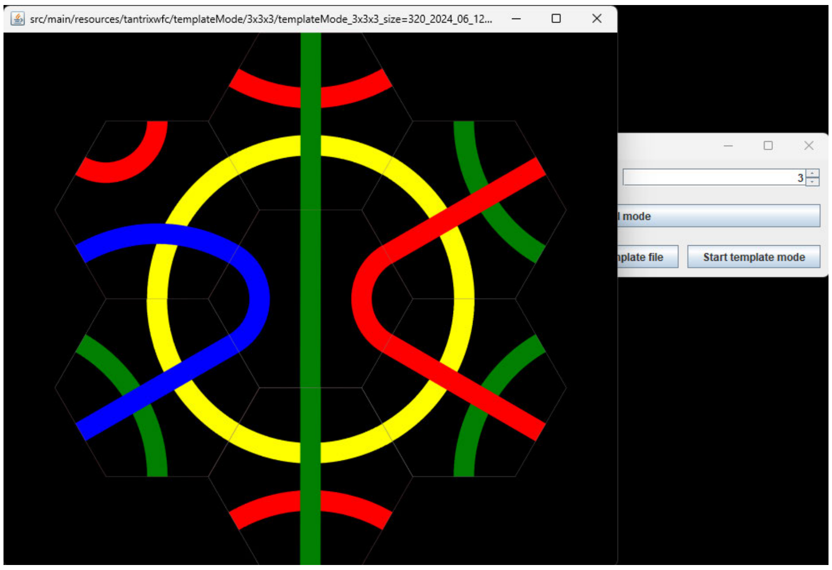
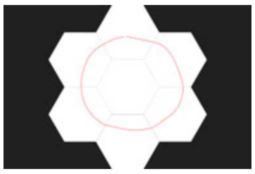
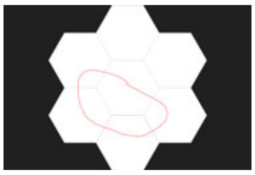
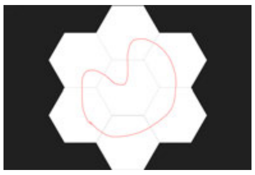
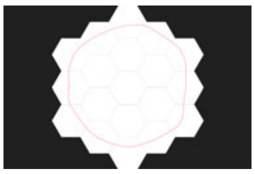

src/main/resources/tantrixwfc/templateMode/3x3x3/templateMode_3x3x3_size=320_2024_06_12...
3
l mode
nplate file
Start template mode

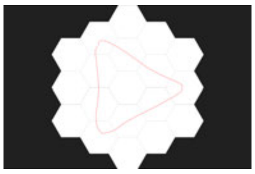
Figure 6.8: The output image is automatically displayed once the algorithm is finished.

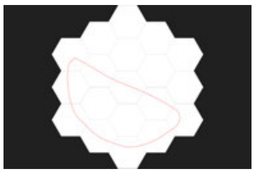

(a) 3x3x3_easy

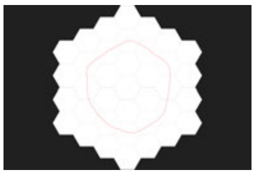

(b) 3x3x3_medium

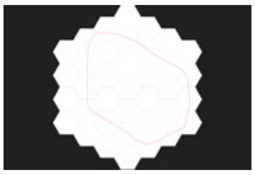

(c) 3x3x3_hard

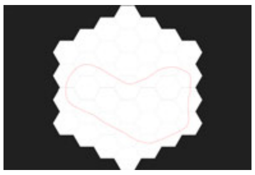

(d) 5x5x5_easy

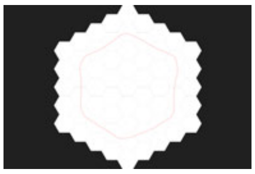

(e) 5x5x5_medium

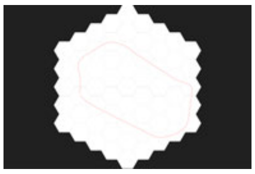

(f) 5x5x5_hard

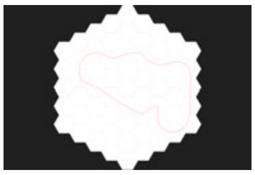

(g) 7x7x7_easy

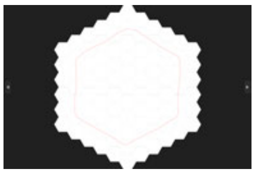

(h) 7x7x7_medium

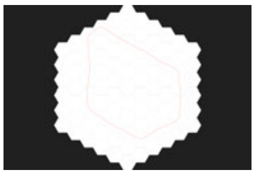

(i) 7x7x7_hard

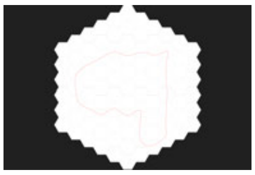

(j) 9x9x9_easy

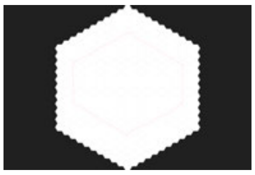

(k) 9x9x9_medium

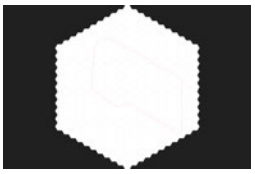

(l) 9x9x9_hard


(m) 11x11x11_easy


(n) 11x11x11_medium


(o) 11x11x11_hard


(p) 21x21x21_easy


(q) 21x21x21_medium

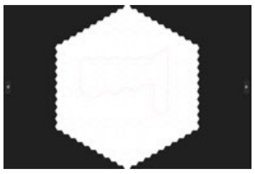
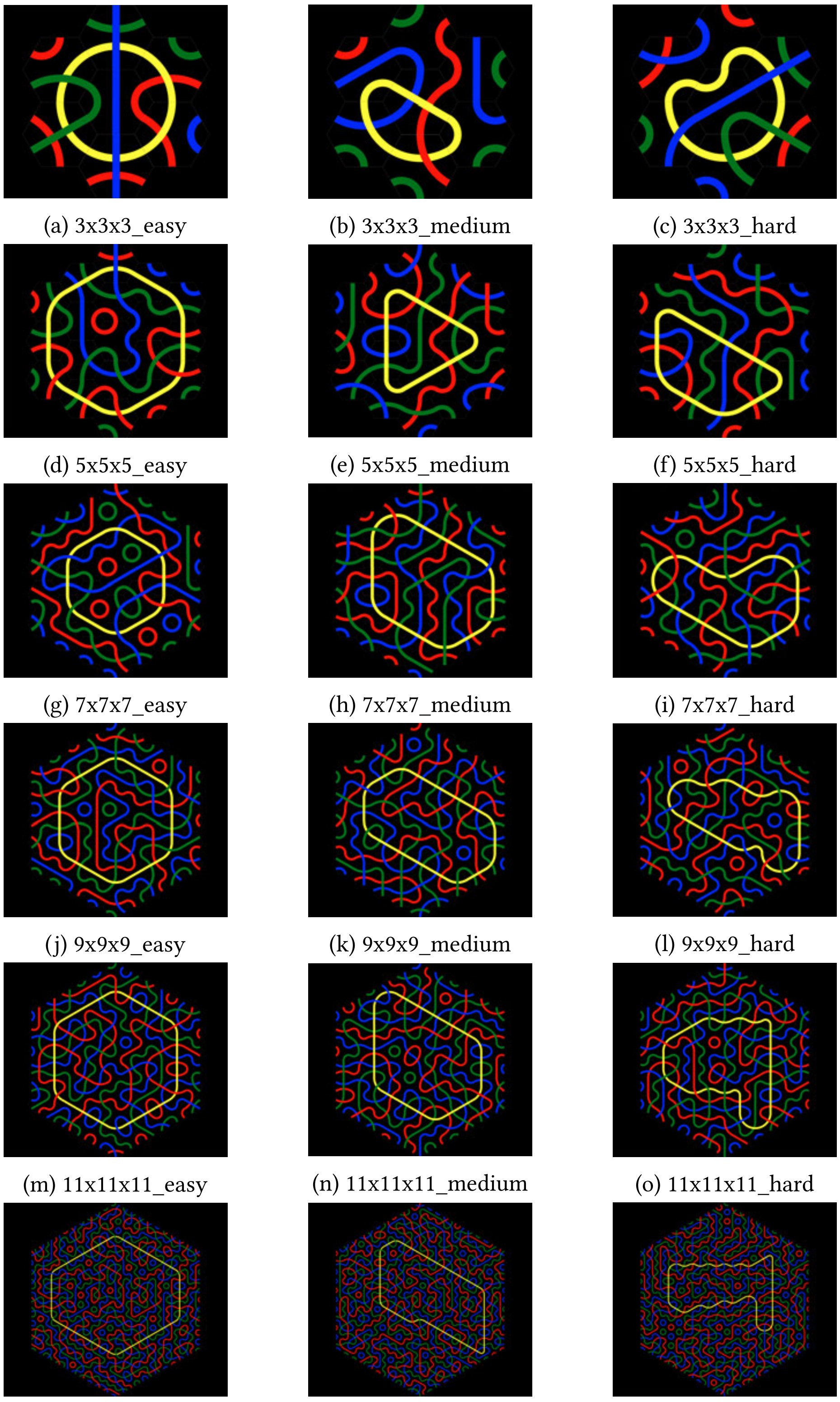

(r) 21x21x21_hard

Figure 6.9: Filled out template files of different grid sizes with different difficulties.



(a) 3x3x3_easy (b) 3x3x3_medium (c) 3x3x3_hard

(d) 5x5x5_easy (e) 5x5x5_medium (f) 5x5x5_hard

(g) 7x7x7_easy (h) 7x7x7_medium (i) 7x7x7_hard

(j) 9x9x9_easy (k) 9x9x9_medium (l) 9x9x9_hard

(m) 11x11x11_easy (n) 11x11x11_medium (o) 11x11x11_hard

(p) 21x21x21_easy (q) 21x21x21_medium (r) 21x21x21_hard

Figure 6.10: Results of the *template mode* for templates of different grid sizes and difficulties.

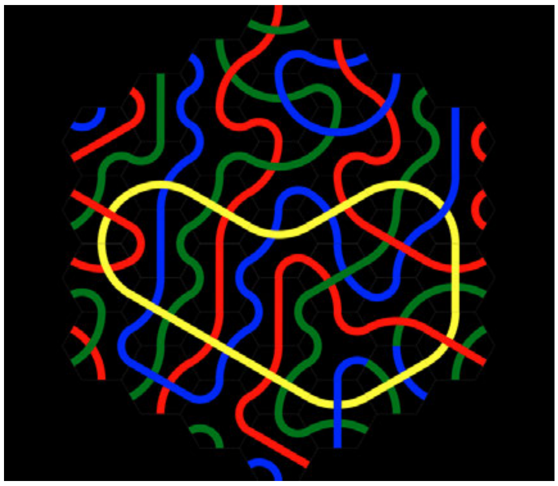


Figure 6.11: False output image that should not have been created during *template mode*.

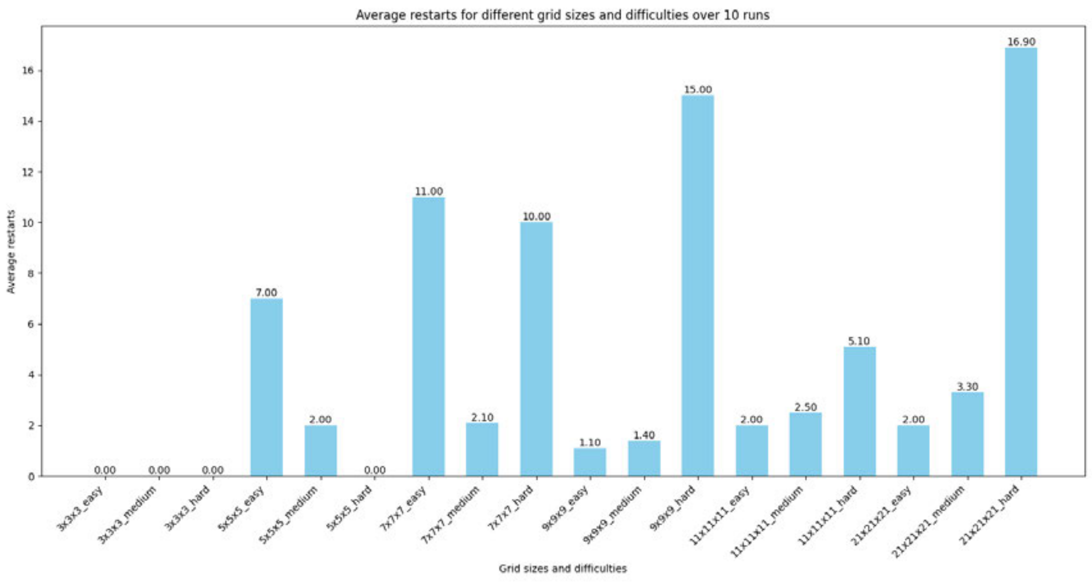


Figure 6.12: Average amount of restarts for different grid sizes and difficulties in *template mode*.

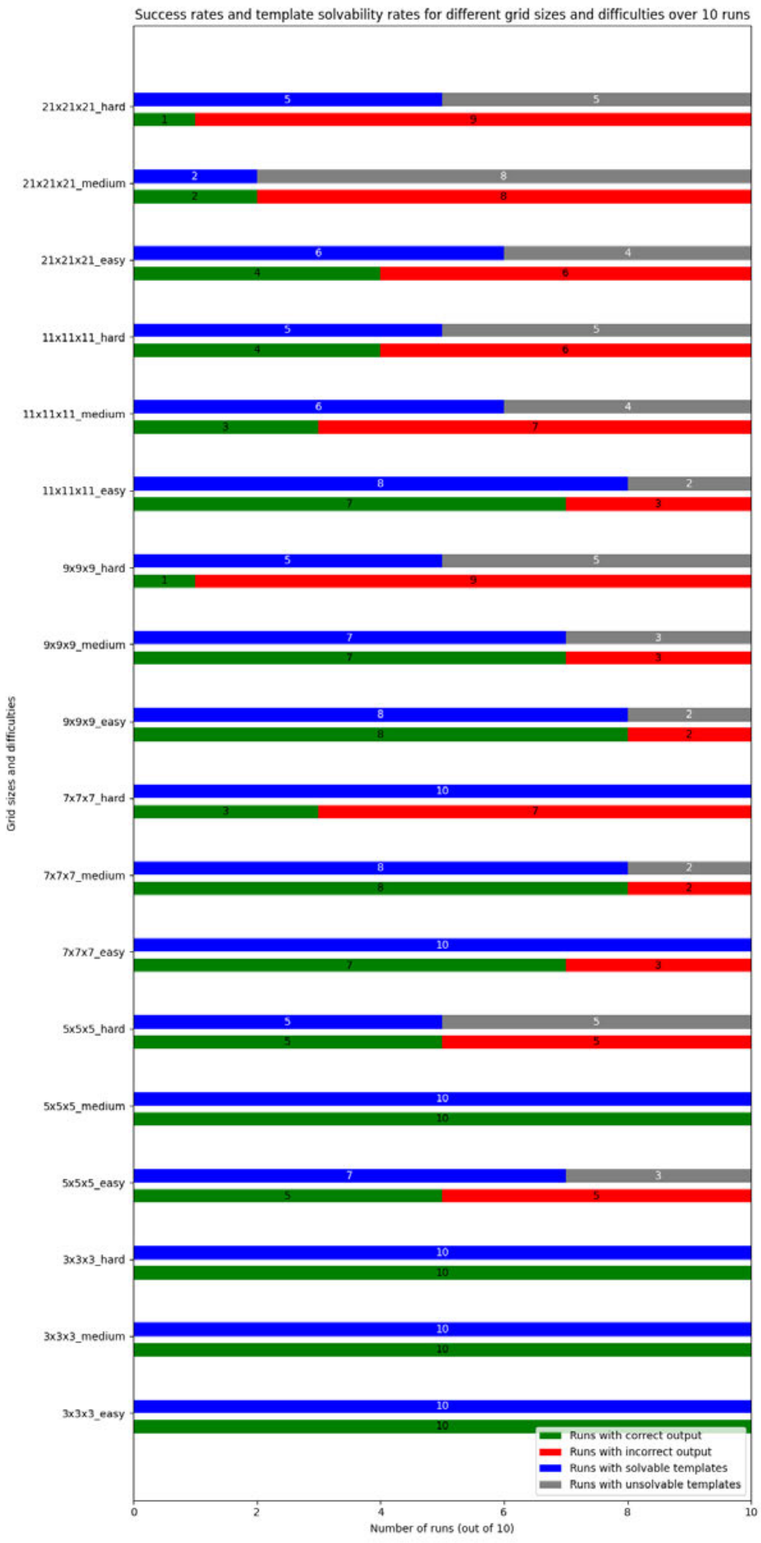


Figure 6.13: Success rates and template solvability rates for different grid sizes and difficulties in *template mode*.

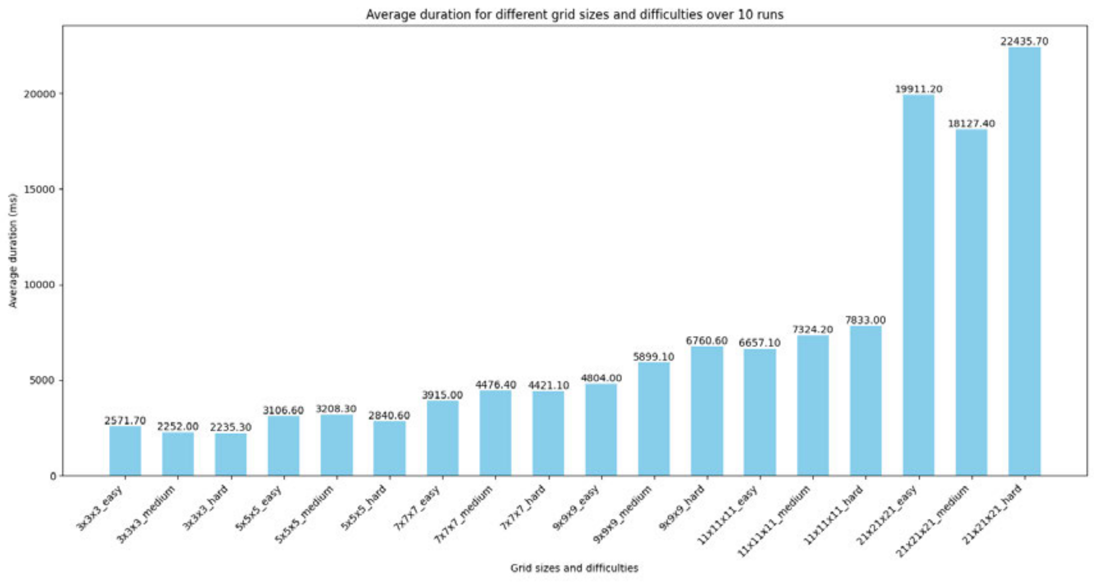


Figure 6.14: Average durations in *template mode* for different grid sizes and difficulties.

7 Conclusion

7.1 Summary

This work creates a digital representation of the *Tantrix* board game and uses the Wave Function Collapse algorithm to solve the game.
As a first preparation step, the game tiles from the real world are digitally recreated, yielding a PNG image file for every distinct tile. Next, the constraint file, which describes what tiles can be adjacent to each other, is built. For both steps Python scripts are being used.
The main software itself is divided into two main parts: the *normal mode* and the *template mode*. The *normal mode* works like the *Simple Tiled Mode* introduced by Maxim Gumin (Gumin [2022]). A grid of a user defined size, containing only empty cells at the beginning, is sequentially filled with tiles. Each tile that is placed adheres to its adjacency constraints. The completely filled grid is finally saved to a PNG file as the output. In order for both modes to work, the framework, that this work is based on, needed to be expanded to support hexagonal tiles, instead of the original squared format. Additionally, due to the inherent restrictiveness of the *Tantrix* tiles, before a tile is placed on the grid, a neighborhood check has to be done to avoid creating unsolvable situations.
A custom GUI allows the user to enter the desired grid size and start both modes. Furthermore, the user can instruct the creation of an empty template file which is needed for the *template mode*. This file contains a grid of empty hexagons in which the user can draw a shape that the algorithm then tries to recreate. The filled out file can be loaded into the program via the GUI, prior to starting the *template mode*.
Before the *template mode* starts the normal WFC, it tries to imitate the hand drawn shape, using tiles with yellow lines. In order for the shape that is generated this way to be solvable, each color that crosses this yellow shape needs to be present in an even amount of lines. The *template mode* possesses a simple mechanism to try to create such a color distribution. In case the algorithm is faced with contradictions, it restarts. If the amount of restarts reaches a predefined threshold, this indicates that the shape might be unsolvable due to the current color distribution. As a consequence, the template file is retraced in a new attempt to create a

solvable shape.
During evaluation of the algorithm, several problems became apparent. The *normal mode* works well for smaller grid sizes, creating no contradictions and therefore not needing any restarts. However, at grid sizes of 31x31x31 and above, the program starts to run out of heap space, causing it to cancel the execution.
In the *template mode*, there seems to be a bug which sometimes hinders the algorithm from identifying contradictions. This prevents the initiation of required restarts and causes the algorithm to place inappropriate tiles. The result is a faulty output image.
Nevertheless, in general, the algorithm is able to correctly retrace the input shape and produce valid output images. That was the aim of this work, which is why the implementation can be considered to be a success. The impact of the template processing on the runtime is noticeable when comparing the *template mode* to the *normal mode*. However, the maximum difference in average runtimes for the same grid sizes between the two modes was only about 8 seconds, which is considered acceptable.

7.2 Outlook

While the implementation already works in its current state, there is still room for improvements. The first (obvious) task for the future is to find the bug that sometimes prevents the identification of contradictions. Fixing this should elevate the success rate of the algorithm to 100%, since then restarts of the algorithm would be triggered correctly until a correct output is generated.
Furthermore, in regard to memory usage, the algorithm can certainly be improved. During execution, it might be advantageous to use other data types than BufferedImages for the tiles, or to optimize the processing/handling of those.
In general, the algorithm could feature a check if the shape provided by the user is solvable at all. Additionally, after the shape is retraced by the algorithm, it could already examine the color distribution of the shape for solvability before handing it to the normal WFC part of the program. This could eradicate unnecessary restarts later on.
While the restart mechanism should guarantee valid outputs, this might come at a severe time cost. There are already implementations of the WFC that have the ability of backtracking, should a contradiction be encountered. Introducing this feature to the current implementation should also improve the runtime.
Lastly, this work only uses *Tantrix* tiles that exist in the real version of the game. It would

be interesting to introduce tiles to the program with color and line combinations that are not available in the original version.

Bibliography

Adel Alshamrani and Abdullah Bahattab. A comparison between three sdlc models waterfall model, spiral model, and incremental/iterative model. *International Journal of Computer Science Issues (IJCSI)*, 12(1), 2015.

Marios C Angelides and Harry Agius. *Handbook of digital games*. John Wiley & Sons, 2014.

K. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003. ISBN 9781139438704.

A.A. Efros and T.K. Leung. Texture synthesis by non-parametric sampling. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, 1999. doi: 10.1109/ICCV.1999.790383.

Mahbouba Gharbi, Arne Koschel, Andreas Rausch, and Gernot Starke. *Basiswissen für Softwarearchitekten: Aus-und Weiterbildung nach iSAQB-Standard zum Certified Professional for Software Architecture–Foundation Level*. dpunkt. verlag, 2018.

Maxim Gumin. Wavefunctioncollapse. https://github.com/mxgmn/WaveFunctionCollapse, 2022. Accessed: 03/06/2024.

G. Gutin and A. P. Punnen. *The Traveling Salesman Problem and Its Variations*. Springer, 2007.

Petra Hofstedt and Armin Wolf. *Einführung in die Constraint-Programmierung. Grundlagen, Methoden, Sprachen, Anwendungen*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007.

J. Jaffar and J.-L. Lassez. Constraint logic programming. Association for Computing Machinery, 1987.

Achim Janser and Wolfram Luther. Der bresenham-algorithmus und andere graphische grundprozeduren. In *Multimedia und Computeranwendungen in der Lehre*. Springer Berlin Heidelberg, 1992. ISBN 978-3-662-00998-7.

Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8, 1977.

Paul Merrell. Example-based model synthesis. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936288.

Niloy J. Mitra, Iasonas Kokkinos, Paul Guerrero, Nils Thuerey, Vladimir Kim, and Leonidas Guibas. Creativeai: deep learning for graphics. In *ACM SIGGRAPH 2019 Courses*, SIGGRAPH '19, New York, NY, USA, 2019. Association for Computing Machinery.

Roger Mohr and Thomas C. Henderson. Arc and path consistency revisited. 1985.

Klaus Pohl and Chris Rupp. Basiswissen requirements engineering–aus und weiterbildung nach ireb standard zum certified professional for requirements engineering foundation level, 4. *Auflage, Heidelberg, Dpunkt. verlag*, 2015.

Francesca Rossi, Peter van Beek, and Toby Walsh. *Chapter 4 Constraint Programming. In: Handbook of Knowledge Representation, Bd.3.* Elsevier, 2008.

Arunpreet Sandhu, Zeyuan Chen, and Joshua McCoy. Enhancing wave function collapse with design-level constraints. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450372176.

Noor Shaker, Julian Togelius, and Mark Nelson. *Procedural Content Generation in Games.* 2016. ISBN 978-3-319-42714-0. doi: 10.1007/978-3-319-42716-4.

Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. Procedural content generation via machine learning (pcgml). *IEEE Transactions on Games*, PP, 2017. doi: 10.1109/TG.2018.2846639.

Julian Togelius, Emil Kastbjerg, David Schedl, and Georgios N. Yannakakis. What is procedural content generation? mario on the borderline. PCGames '11. Association for Computing Machinery, 2011.

Li-Yi Wei, Sylvain Lefebvre, Vivek Kwatra, and Greg Turk. State of the art in example-based texture synthesis. In *Eurographics 2009, State of the Art Report*, 2009.

Glossary

Adjacency constraint	A rule that dictates, which tile can be next to another tile.
Cell	A single unit within the wave in the WFC, which shall be collapsed to a single state by the algorithm. Also one unit within the grid that is filled by the WFC.
Corner	A line on a *Tantrix* tile that connects two adjacent edges.
CSP	Constraint Satisfaction Problem - defines a problem, which can be solved with a constraint solver.
Curve	A line on a *Tantrix* tile that connects two edges, which have one edge in between them.
Grid	A combination of multiple cell, arranged in a coordinate system.
Normal mode	Mode that implements the simple tiled model from the WFC, adapted to support hexagonal tiles.
PCG	Procedural Content Generation - used to automatically generate assets, for example buildings in a computer game.
Shape	The outline of the drawing, that a user provides during *template mode*.
Straight	A line on a *Tantrix* tile that connects two edges opposite from one another.
Tantrix	The board game that this thesis is based on.
Template file	A PNG file that contains a hand drawn shape which should be recreated in the *template mode*.
Template mode	Mode that extends the *normal mode* by adding the ability to create and process a template file.
Tile	Represents a single state that a cell can be collapsed to. It also contains the image that is shown on the grid for the cell, which was collapsed to this tile.

Tracing	The recreation of a hand drawn shape, using the yellow lines on the *Tantrix* tiles.
WFC	The Wave Function Collapse algorithm, devised by Maxim Gumin.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 18. Juni 2024 [redacted]