Combining Wave Function Collapse and Evolutionary Algorithms for Controlled Content Generation

Ioannis Brellas
Institute of Digital Games
University of Malta

A thesis submitted for the degree of
Master of Science in Digital Games

June, 2021

University of Malta Library – Electronic Thesis & Dissertations (ETD) Repository

The copyright of this thesis/dissertation belongs to the author. The author's rights in respect of this work are as defined by the Copyright Act (Chapter 415) of the Laws of Malta or as modified by any successive legislation.

Users may access this full-text thesis/dissertation and can make use of the information contained in accordance with the Copyright Act provided that the author must be properly acknowledged. Further distribution or reproduction in any format is prohibited without the prior permission of the copyright holder.

Abstract

Wave Function Collapse (WFC) is a procedural content generation algorithm introduced by Maxim Gumin that has risen in popularity in recent years. The Wave Function Collapse algorithm utilizes two distinct models, the Overlapping and the Simple Tiled model, to divide input bitmap-based or tiled-based images into patterns of various shapes and consistently generate output images of a larger scale, which feature the same patterns. Although WFC is able to create images that are visually stunning in massive numbers, there hasn't been many attempts to generalize it, in order to make it able to be applied for game content generation. Specifically, the images that are generated can be used as textures for games and sometimes as levels, but the playability of these levels is not guaranteed. In this thesis, WFC is combined with an evolutionary algorithm in an attempt to control the generated outputs of WFC and push them towards a more playable nature. The implemented algorithm evolves patterns, in the form of tiled images, that will then be used as input for the WFC algorithm. The idea is to first create visually flawless images through an evolutionary procedure, using a given tileset, that will result in the generation of similarly flawless images and then elaborate further so that some control over the generated content of the WFC algorithm is established. First, we go through every parameter of our evolutionary algorithm, like the selection method, the fitness function, the genetic operators and the population size, exploring the impact that each of them can have on the produced results and then we propose an optimal setup for our approach. The proposed setup managed to have a very promising performance and within a reasonable amount of computational resources. Furthermore, the algorithm managed to maintain the same performance when tested on totally random tilesets, while the results that were being produced through the WFC algorithm were increasing in complexity for larger tilesets, while maintaining the algorithm's wide expressive range. Finally, despite the good performance of our implementation, there is definitely some room for improvement. In this approach we explored many of the parameters that can have an impact on the evolution of the input patterns, but there is still much research to be done in determining the best approach. Alternatively, this approach can be utilized by future implementations that want to address similar problems.

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr Antonios Liapis, for his continuous and invaluable guidance and involvement during the whole duration of my dissertation.

Then, I would like to thank all the alumni and staff of the IDG department for the knowledge that they selflessly shared with me and their immediate help when asked for it. Furthermore, I would like to thank my friends, the ones I had and the ones I made along the way, for being there and making my day when I needed it the most.

Last but not least, I would like to thank my family for their immeasurable support through all these years. Thanks for being awesome.

Statement of Originality

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Ioannis Brellas

Contents

1 Introduction **1**
1.1 Brief Description of the Problem 1
1.2 Challenge 2
1.3 Research Question 2
1.4 Contribution 3
1.4.1 Evolution and Wave Function Collapse 4
1.4.2 Simplifying the Input of the Simple Tiled model of the Wave Function Collapse Algorithm 4
1.5 Thesis Outline 5

2 Related Work **7**
2.1 Procedural Content Generation 7
2.1.1 Computational Creativity Facets 7
2.1.2 Why use Procedural Content Generation in Games? 8
2.1.3 Procedural Content Generation Methods 9
2.2 Wave Function Collapse 11
2.2.1 Overlapping Model 11
2.2.2 Tiled Model 13
2.2.3 3D Wave Function Collapse 15
2.2.4 Wave Function Collapse Applications 15
2.3 Evolutionary Algorithms 17

3 Methodology **21**
3.1 Wave Function Collapse Components 21
3.1.1 Tileset 21
3.1.2 Ruleset 22
3.1.3 Wave Function Collapse Setup Summary 23
3.2 Evolutionary Algorithm 23
3.2.1 Population 23
3.2.2 Gene Selection Method 24
3.2.3 Fitness Function 24
3.2.4 Genetic Operators 25
3.2.5 Replacement Method 27

4 Experiments **29**
4.1 Performance Metrics 29
4.2 Experimental Setup 31

4.3 Genetic Operators Testing 31
4.4 Population Count Testing 35
4.5 Resolution Testing 37
4.6 Fitness Function Testing 38
4.7 Expressivity Analysis 41
4.8 Robustness 44
4.8.1 Setup 45
4.8.2 Results 46

5 Conclusion 49
5.1 Results summary 49
5.2 Addressing the research question 51
5.3 Limitations 53
5.3.1 Tileset 53
5.3.2 Ruleset generation 54
5.3.3 Simple Tiled model 55
5.4 Future Work 55
5.4.1 Automated symmetry system 55
5.4.2 Alternative applications of evolution on Wave Function Collapse . . 55
5.4.3 Mixed Initiative Approaches 56
5.4.4 Going 3D 56

List of Figures

2.1 The six facets of computational game creativity. 9

2.2 Examples of various bitmap inputs and their outputs using the overlapping model WFC. Source: Maxim Gumin's github [8] 12

2.3 An example tileset and the tilemap generated via the Simple Tiled WFC model. Source: Maxim Gumin's github [8] 13

2.4 Two examples of applying crossover on a binary chromosome. Red and blue genes represent the two different offspring emerged from each crossover operator. Source: Game AI Book [42] 18

2.5 Two examples of mutating a binary chromosome. In each picture the top chromosome is the selected one and the bottom chromosome is the mutated one. Source: Game AI Book [42] 18

3.1 Three examples for the phenotype images of the randomly initialized population for the simple (left), medium (middle) and full (right) tileset. 24

3.2 Visual representation of pixel similarity calculation. The tile highlighted by the red square is the tile for which we calculate the pixel similarity, while the blue lines indicate which edge pixels are taken into account from the adjacent tiles. 25

4.1 Two image of different KL divergence values. On the left we have an image with a KL divergence of 0.53 and on the right one with a KL divergence of 1.81 (same tileset). 29

4.2 Visual representation of the Von Neumann (left) and the Moore (right) neighborhood removal techniques. The chosen tile is highlighted in blue, while its neighboring tiles are highlighted in red. 33

4.3 The average values (across 40 runs) for the final fitness and the KL divergence for each of the operators on all three tilesets. 35

4.4 Final fitness average values (across 40 runs) for each of the population sizes on all three tilesets. 37

4.5 Final fitness average values (across 40 runs) for each resolution on all three tilesets (resolution numbers refer to tiles not pixels). 39

4.6 Visual representation of the paths that are calculated. The blue lines are the paths that have been found by the A* algorithm, while the houses that are highlighted in red are the ones that are not connected to any other house. . 40

4.7 Our algorithm's expressive range in terms of number of colors and compressed complexity for each of the tilesets. 43

4.8 The generated images that feature the minimum (up) and maximum (down) values for the number of distinct colors and the compressed complexity for the simple (left four), medium (middle four) and full (right four) tilesets. The images on the left of each quartet correspond to the size, while the right correspond to the colors. 44
4.9 Two examples of generated images for each of the five tested tilesets, starting from tileset 1 on the left and ending at tileset 5 on the right. 47

5.1 Two tiles that have slight dissimilarities on their edges. 53
5.2 Examples of tiles that have some similar pixels but do not fit each other. . . 54
5.3 Two examples of tiles that can't be assigned a symmetry type. 54

List of Tables

2.1 The rotational changes of the tiles according to their symmetry type. 14

3.1 The number of tiles for each tileset. 22
3.2 All the genetic operators and their corresponding methods that were used to mutate our genes. 27

4.1 Average Values (over 40 runs) for each of the Genetic Operator for the Simple Tileset.(The numbers is brackets are the confidence intervals of the corresponding metrics). 33
4.2 Average Values (over 40 runs) for each of the Genetic Operator for the Medium Tileset. (The numbers is brackets are the confidence intervals of the corresponding metrics). 33
4.3 Average Values (over 40 runs) for each of the Genetic Operator for the Full Tileset. (The numbers is brackets are the confidence intervals of the corresponding metrics). 34
4.4 Average Values (over 40 runs) for each population size for the Simple Tileset. (The numbers is brackets are the confidence intervals of the corresponding metrics). 36
4.5 Average Values (over 40 runs) for each population size for the Medium Tileset. (The numbers is brackets are the confidence intervals of the corresponding metrics). 36
4.6 Average Values (over 40 runs) for each population size for the Full Tileset. (The numbers is brackets are the confidence intervals of the corresponding metrics). 36
4.7 Average Values (over 40 runs) for each distinct resolution for the Simple Tileset. (The numbers is brackets are the confidence intervals of the corresponding metrics). 38
4.8 Average Values (over 40 runs) for each distinct resolution for the Medium Tileset. (The numbers is brackets are the confidence intervals of the corresponding metrics). 38
4.9 Average Values (over 40 runs) for each distinct resolution for the Full Tileset. (The numbers is brackets are the confidence intervals of the corresponding metrics). 38
4.10 Average Values (over 40 runs) for each fitness function on the Medium and the Full Tilesets. (The numbers is brackets are the confidence intervals of the corresponding metrics). 40
4.11 Details about each of the tilesets used for the robustness testing. 45

4.12 Average Values (over 40 runs) for each of the tilesets that were used for the robustness testing.(The numbers is brackets are the confidence intervals of the corresponding metrics). 46

Chapter 1

Introduction

In this chapter a brief description of the problem that this thesis will try to tackle is going to be presented. Then some of the challenges that this problem poses along with the formulated research question are going to be discussed. Finally, the contributions of this research will be explained and an outline of the rest of the work will be given.

1.1 Brief Description of the Problem

One of the 6 game facets of game content [21] and one of the most important parts of digital games is the levels. Levels in games range from simple labyrinth-like stages to vast 3D worlds or even endless terrain that the player can travel through and explore. The generation of levels is a task whose difficulty ranges accordingly. To be more specific, the generation of a simple level is a relatively easy task that can be quickly and efficiently done by a human being. However, the creation of a huge 3D environment, consisting of hundreds of different elements (trees, houses, mountains etc.) or the generation of hundreds of simple levels are tasks that requires the cooperation of multiple game developers.

This is one of the reasons why procedural content generation has become and is still becoming more and more popular in recent years. Procedural Content Generation, or PCG in short, refers to the automated creation of game content, such as levels, using a computer. Specifically, a computer can be programmed in such a way that it can massively produce any type of game content that a game might require and most importantly it can also do it faster than a human developer. PCG can be used by game developers as an assisting tool to help them carry out the task of generating game levels and can also provide them with ways to tackle issues like novelty or replayability.

A very interesting PCG algorithm that has been rising in popularity recently is that of Wave Function Collapse, as it was introduced by Maxim Gumin [8]. Gumin, inspired by quantum mechanics, implemented a PCG algorithm that could massively and quickly generate content using smaller samples of the content that needed to be generated as input. The content that is most commonly generated is 2D images, but there has been applications and games that generate 3D content as well. The Wave Function Collapse (WFC) algorithm uses two distinct models to do that. The first one is called the Overlapping model and it uses a small bitmap image as input to generate bitmaps of larger scale that feature the same patterns as the input image. The second one is the Simple Tiled model and in this case the input of the algorithm is a set of tiles and a set of adjacency rules for them. In this case, the algorithm is turned into a constraint solving algorithm that generates a tilemap

with respect to the ruleset that is given to it as input. The generational value of the WFC algorithm comes from its speed and its ability to create new content by simply feeding a different random seed to the algorithm.

WFC is a very fast and efficient PCG algorithm and the generated results are most of the time astonishing in terms of aesthetic quality. However, there are still issues that make it hard to be applied as a level generation tool. The most important of these issues is the randomness that comes with it. Although the images that are produced are aesthetically pleasing, they can not always be used as game levels, the reason being that they might not be playable. This means that the algorithm on its own is not very reliable to be used as a game level generation tool and we either have to enforce some game design specific constraints on it or we have to manipulate some of its core parameters, like the inputs or the core generative procedure, in order to ensure that the content that is being generated can be useful for game level designers.

1.2 Challenge

The way that this thesis will address the aforementioned problem is by combining WFC with an evolutionary algorithm. The final goal is to implement an application that can be used by game level designers, as a helping tool during the development of a game. The idea is that designers could utilize this application to gain some control over the levels that are being produced by the WFC algorithm, without having to invest much time on it.

This task proved to be a bit more complicated than what initially expected, due to some challenges that arose along the way. The most important one was to implement an application that could be used by designers universally, meaning that it shouldn't be bound to specific inputs, like for example one specific set of tiles or only images that can be used for 2D adventure games. This means that a way that could work with any given input and still perform relatively well and in acceptable amounts of time had to be introduced. Combine with the fact that the WFC algorithm is highly tied to the input that is given to it, meaning that a lot of times this input can be rejected, in the sense that no output is being produced though it, it adds up even more to the complexity of the problem.

The second most important challenge was applying the evolutionary algorithm on the WFC algorithm. WFC is slightly delicate as an algorithm when it comes to modifying it, meaning that slightly changing the wrong parameter might break the whole procedure. Thus, when picking a parameter to evolve one that will not cause major issues, but at the same time will provide good enough results, has to be picked. At the same time, the WFC algorithm has a wide expressive range when it comes to the content that is generated. Combining it with another algorithm might have an impact on the expressivity of the algorithm, so another challenge would be to preserve the expressive range of WFC and not negatively impact it, when combining it with an evolutionary algorithm or any other algorithm.

1.3 Research Question

Within the context of this thesis, an effort to expand on the WFC algorithm and specifically to manipulate it in such a way that the generated content is not just aesthetically pleasing, but also playable, will be attempted. To do that an evolutionary algorithms will be used to evolve a set of input patterns that can then be used to produce images that may potentially

feature the intended patterns on them too. Specifically, the main goal is to implement an application, or at least the core functionality of it, that will enable game designers to generate content using the WFC algorithm, by simply providing a tileset of their choice.

From that point of view, the research question consists of the following four parts:

- **Can evolutionary algorithms be applied to the WFC algorithm in order to control the stochasticity of the outputs produced by it, without requiring many computational resources?** This is the core part of this research. Basically, it refers to the degree of efficiency of the chosen approach, which is measured in terms of performance versus required computational time. Most of the experimentation phase is dedicated to testing and evaluating this part.
- **Which parameters are the most important in determining an optimal setup for the evolutionary algorithm?** This question refers to the various parts that make an evolutionary algorithm, whether those are the gene representation, the number of generations or even the genetic operator used. These parameters have to be established through separate experimentation, which will give us the optimal setup for this approach when combined.
- **Are the results provided using this method diverse?** This part refers to the ability of the algorithm to produce results that are different from each other and use the resources given in an optimal way. It can be measured through a visual representation of the expressive range of the algorithm, or by establishing a way to calculate the percentage of input patterns that are being used on the generated content over all the available input patterns.
- **Is the evolutionary algorithm robust enough to work on any given tileset?** In this part we are looking to examine if the algorithm can work on a universal basis and produce results using any given tileset. This will ensure that this algorithm can be used as a tool for game designers and the way we evaluate it is through its performance when given random inputs (random being those that we didn't specifically create for this task).

Each part of the research question is addressed in this thesis. On every step each of the possible options are carefully examined and the reasoning behind our decisions and assumptions is explained, based on the results that are produced. At the same time, the computational time of our algorithm, as well as, its expressive capabilities are taken into consideration, before rushing into the final speculations.

1.4 Contribution

Addressing the research question proved to be a more challenging task than expected at the beginning. Wave Function Collapse might seem like a simple procedure at first glance, but gets more complicated the deeper you dive into it. Despite these drawbacks, the implementation managed to perform very well and managed to address the research question relatively successfully. In the process of doing so, this approach offered some contributions when it comes to expanding on the WFC algorithm, that may be used as stepping stones for further research or applied as solutions to similar problems. The following sections present an overview of these contributions:

1.4.1 Evolution and Wave Function Collapse

During the past years there has been some research on controlling the outcome of Wave Function Collapse, especially when it comes to developing games that use it. The developers of such games had to enforce extra constraints on the algorithm to ensure that the content that is generated can be used as content in them. However, most of these approaches are based on either using game specific constraints [32] or pre-propagating [26] some parts of the output to ensure that some the playability of the level stays intact.

There haven't been many approaches [17] where an evolutionary algorithm is used in combination with the Wave Function Collapse in order to control the results that are being produced. In this thesis, an extended research on how an evolutionary algorithm can be utilized to evolve the elements that can be used as input so that the results that the WFC algorithm generates are usable as game levels is going to be provided.

Specifically, this approach will focus on evolving only the input parts of the WFC algorithm, but during this process the parameters that are the optimal for doing so will be examined. By doing this evolutionary parameters are the most important in contributing to the efficiency of the algorithm will be discovered, as well as the parts of the WFC algorithm that could be evolved and provide a better outcome than ours will be explored.

1.4.2 Simplifying the Input of the Simple Tiled model of the Wave Function Collapse Algorithm

Although for the WFC algorithm the input that is used on the Overlapping model is relatively simple to create, the case is not the same when it comes to the Simple Tiled model. For the latter, the user has to provide the algorithm with a tileset accompanied by a set of rules. Both of these have to be manually created by the user or the user has to come up with a custom algorithm to automatically craft those. Creating the tileset can be a simple task, but creating the ruleset can be quite a challenge, especially for bigger tilesets. To create the ruleset the user has to go through all the tiles and establish both the symmetry type of the tile and the adjacency rules that tie it with the rest of the tiles in the tileset.

There have already been attempts to automate the generation of the ruleset by Joseph Parker [29], who implemented an application that only requires the user to assign a symmetry type to each of the tiles and then draw a sample image that will be used to generate the ruleset. Although this simplifies the procedure a lot, the user still has to draw a sample input image. This still requires the user to participate in the procedure of content generation and takes away from some of the novelty of the procedural generation, since the image that is drawn will be closer to some of the already existing content that the user is familiar with.

That is why in this approach, the main functionality of the evolutionary algorithm will be to produce these sample input images/patterns, which can then be combined with Parker's [29] implementation to generate the rulesets that are needed for the WFC algorithm. This way the only thing that the user has to provide is the symmetry type for each of the tiles. This can still be a challenging task especially for large tilesets, but the workload is definitely lower than before and the fact that the algorithm generates the input image on its own, adds up to the novelty of the results.

1.5 Thesis Outline

In Chapter 1 (Introduction) a brief description of the problem that will be adressed, as well as, the research question and the main contributions of this thesis were presented. In Chapter 2 (Related Work) the basics of Procedural Content Generation will be discussed, a more detailed description of the Wave Function Collapse algorithm will be given, some of the fundamentals of Evolutionary Algorithms will be explained and some interesting applications of the WFC algorithm will be featured. Then, in Chapter 3 (Methodology) all the tools that were implemented in order to address the research question will be introduced. In Chapter 4 (Experiments) the experiments that were conducted throughout this thesis will be presented and their results will be discussed. Finally, in Chapter 5 (Conclusion) a summary of the results is given, the limitations that were encountered during this approach will be discussed and suggestions on some improvements that could be done as future work to expand on this thesis will be proposed.

Chapter 2

Related Work

In this Chapter a brief overview of the components, the technologies and the algorithms that were used during the implementation process of this thesis is going to be presented, in order for the readers to further understand and comprehend each part of the development process. The topics that are going to be discussed and which are crucial to understanding this thesis are Procedural Content Generation, the Wave Function Collapse algorithm and Evolutionary Algorithms. Furthermore, a few previous approaches on Wave Function Collapse as well as some applications of the algorithm are going to be reviewed.

2.1 Procedural Content Generation

Procedural Content Generation is one of the most popular areas of Game AI and has risen in popularity in the recent years. When we talk about Procedural Content Generation - also seen as the PCG abbreviation - we refer to *the algorithmic creation of game content with limited or indirect user input* [35]. In other words, PCG is the automatic generation of content that can be used in games by either a computer on its own or together with one or more human designers.

2.1.1 Computational Creativity Facets

Game content can refer to anything that can appear in a single video game ranging from maps, levels, graphics, music, textures, rules, characters, weapons etc [35, 42]. However, there are some parts of a game that are not considered to be content. Such examples are the game engine as well as the NPC behavior. However, even when these two are excluded there are still plenty of elements that can be considered game content. That is why Liapis et al. [21, 20] categorised the game contents into six creative facets: visuals, audio, narrative, levels, rules and gameplay.

- *Visuals*: digital games are most commonly rendered on a screen and thus they heavily rely on their visual output. The visual effects of a game can range from photorealistic, to caricaturized, to abstract. Photorealistic refer to realistic representations of real-life objects or sometimes when that is not possible, like for example in science fiction or fantasy games, to the real-world reference material that is then altered in a way to satisfy the fictional environment they belong to. Meanwhile, caricaturized visuals focus on bringing out specific emotions and abstract visuals refer to cases where artists are constrained and try to create memorable characters and environments using limited

resources. Some of the most common visuals that can be procedurally generated are plants [10], weapons [7], textures or even particles.
- *Audio*: even though most of the time it is overlooked, audio is a very important part of a game and contributes strongly to the overall experience. Audio is digital games can be every sound that can be heard in the game, from a fully orchestrated soundtracks, like champion theme songs in League of Legends (Riot Games, 2009), to simple sound effects like gun shooting in Counter-Strike: Global Offensive (Valve, 2012), to even simple short sounds that give feedback to the player, like the sound on the victory screen of a game or the sound that plays when the player is damaged. In most cases, the procedural generation of audio is combined with human interaction, indicating that any creativity involved would be combinatorial [6].
- *Narrative*: While narrative in games is a very controversial topic in game academia and many would argue that it is more than just stories, for the sake of simplicity we will define narrative as the extensive storytelling that brings the game together and gives it its unique identity. There has been a lot of research on algorithmic and interactive narrative generation and one of the most notable examples of a game that features procedurally generated narrative is Facade [24] (Procedural Arts, 2005), where the player choices can dynamically change the story of the game.
- *Levels*: levels refer to the virtual space that the game takes place in. It can be the map of a strategy game like Age of Empires 2 (Ensemble Studios, 1999), the 3D world in Sims 3 (EA Games, 2013) or the simple labyrinth-like stages of Pacman (Namco, 1980). Procedural generation of levels is the most popular among the facets [34, 40] and is the one that this thesis will focus on.
- *Rules*: a video game wouldn't be a game if there were no rules. By rules we can either refer to the actual game rules that the player has to follow in order to play the game or the different mechanics [36] of the game. Some of the most generic rules are for example the fact that the player can not move outside of the level's boundaries or the fact that if the player jumps from a very high point in Dark Souls (Namco Bandai Games, 2011) they die to fall damage, while mechanics are for example the jump mechanic in Super Mario Bros (Nintendo, 1985) or the leap of faith mechanic in Assassin's Creed (Ubisoft, 2007).
- *Gameplay*: While the rest of the facets focus on elements that are part of the game itself, the gameplay facet has to do with the overall experience of the player while playing the game. In other words, gameplay refers to the way the player traverses through the virtual world of the game, solves the puzzles or interacts with the NPCs of the game. Simulating such a behaviour is the primary goal of a game AI agent and specifically the focus is on training the agent so that it can learn any type of game, "play" through it and evaluate it in terms of playability, fairness or uniqueness.

2.1.2 Why use Procedural Content Generation in Games?

Content for all the above facets can be created even without the use of computers. However, procedural content generation keeps becoming more and more popular and it turns out that there are plenty of reasons for that. The first reason that almost anyone could think of is the fact that PCG removes the need to have a human designer or artist when developing games [35]. No matter how cruel or sad this might sound, using a computer to do something that humans can do, but faster and most of the times more efficient, is definitely a cheaper option. This combined with the increase of people and time needed for developing a single

game, makes PCG look more appealing to companies that want to find a cheaper and faster solution without giving up on quality at the same time. Also, PCG can enable companies to take more risks when it comes to releasing a game which is"out-of-the-box" and away from the ordinary, since failure translates to lower time and money damage. However, trying to promote PCG by implying that developers and artists will lose their jobs is not a great idea. Thus, the above argument can be shifted around to sound more "acceptable". Specifically, when we talk about PCG in games, we refer to computer augmented content generation [35]. Basically, this means that game developers can use PCG to their advantage to either enhance their already existing content or use PCG as a tool that can provide inspirational assistance. This way, even small teams or even hobbyists can develop games of equal quality to those of a larger company that has access to more resources.

Figure 2.1: The six facets of computational game creativity.

Furthermore, PCG can contribute to creating more novel games. By novelty in games, we refer to what makes the game unique and helps it stand out from the rest of the competition. That can sometimes be its in-game environment, its theme, its mechanics or even its unique characters. Using PCG to create any of the aforementioned content gives the developers a massive advantage in terms of novelty, because a computer has the ability to generate content that might have not been generated in the past, simply because it might not be aware of what was generated in the past. On the same theme, using PCG can make the same game replayable or even infinite, meaning that it can create new game content at the same pace it is "consumed" by the player, providing her with unlimited play time [42].

Another exciting feature of PCG is that it can be tailored in such a way that the content is adapting to the player [30]. This can be done by combining PCG with player modeling, another very popular field in Game AI, so that the content that is generated aims into maximizing the enjoyment of players. Last but not least, PCG can help understand design and creativity. This aligns with the belief, which a lot of computer scientists have, that programmers and scientists do not really understand something until they have to implement it in code. In other words, in order to be able to program a computer to automatically generate game content, the programmer/designer has to be able to know what needs to happen during the design process, as well as, have a grasp of the creative facets that were discussed above.

2.1.3 Procedural Content Generation Methods

There are many algorithms that can be used to procedurally generate content most of which are commonly used in game AI in general. These algorithms can be classified into different categories according to their determinism, controllability, iterativity, autonomy or adaptivity. Below, we are going to discuss some of the most important PCG methods, which are search-based, solver-based and grammar-based methods, as well as, cellular automata, noise and fractals and Machine-Learning based PCG.

- Search-based procedural content generation [41] is the most popular out of the meth-

ods in game academia. Briefly, in this method the content is generated by using an evolutionary algorithm or some other stochastic search or optimization algorithm in order to search for the desired content. The core components of a search-based PCG method are the search algorithm, the content representation and the evaluation function/s. The search algorithm is the core of the method. The most commonly used algorithms are evolutionary ones and their goal is to search through generated data until the derided one is found. The content representation is basically how the content we want to generate is translated into such a form that the algorithm can "understand" and work around. Lastly, the evaluation function is the one that will determine whether what was generated is good enough or not.

- Solver-based methods rely on constraint solvers , like the ones used in logic programming, rather than objective functions. The constraint solver is responsible for searching for the content that satisfies the constraints that are specified in the start. Although, in this case evolutionary algorithms are still a common way to implement a constraint solver, in some cases the problem is reduced to a SAT (satisfiability) problem and a SAT solver is used to find the solution. Another approach is to use Answer Set Programming (ASP) to solve the constraint problem. When using an ASP solver the model (set of parameters) can be used to describe a game world or a story and the constraints can specify playability or some aesthetic considerations.

- Grammar-based methods, as the name suggests, use grammars which are basically sets of production rules to generate content. Grammars are usually used as a constructive method, generating content based on the rules that are defined in them, without having the need for an evaluation function or constraints. However, grammars can also be combined with evolutionary algorithms and provide a genotype-to-phenotype mapping [4].

- Cellular Automata [25] are discrete models of computation that are very popular in fields other than computer science, such as physics, biology or even physical phenomena like growth and development. Their basic idea is very simple: a cellular automaton is a grid of cells that change in time according to some adjacency rules. These rules are used to define the state of a specific cell of the automaton at any given point in time, based on the state of its neighboring cells. Cellular automatas are widely used in PCG because they use a small number of simple parameters and they are relatively easy to understand. Their most common use being terrain generation and especially cave generation [12].

- Noise algorithms as well as fractal algorithms are very frequently used to generate textures and terrain [27]. The main reason for this is that both texture and some aspects of terrains can be represented as two dimensional surfaces. These surfaces are often referred to as maps and in the case of textures they are called intensity maps. Each cell on these maps corresponds to the brightness of its associated pixel and in the case of terrains, where the maps are called heightmaps, each cell corresponds to the height of the terrain at its associated point.

- Last but not least, there has been an increase in approaches that combine PCG with Machine Learning (PCGML) [38]. PCGML refers to the training of various Machine learning models on existing content, so that they are able to generate content of the same type and style. The two most popular methods used for PCGML are generative

adversarial networks [5] and variational autoencoders [15]. Although, using Machine Learning to generate content has proven to be very efficient for some types of content, it is still quite challenging when it comes to game content, because of their advanced complexity as well as the lack of existing content to train the models on.

2.2 Wave Function Collapse

A procedural generation algorithm that has been rising up in popularity recently is that of Wave Function Collapse. Wave Function Collapse as a term was first introduced in quantum mechanics [1]. In this field, the term describes the transition from a superposition of macroscopically distinguishable states to one of them. In other words, the wave function is a representation of the superposition of all the possible states that an unobserved particle can be in and once this particle is observed these possible states disappear and the wave function collapses into one final state. Inspired by this notion, Maxim Gumin implemented an algorithm with the same name [8], that utilises this quantum mechanics theory and uses it to generate content procedurally. Since its release on 30th September 2016, Wave Function Collapse started spreading rapidly among developers and researchers, especially via the use of social media such as Twitter. People would share their experiments using the algorithm and quite a few researchers would jump in to evaluate the algorithm and improve it, while developers would find ways to incorporate it in their games.

2.2.1 Overlapping Model

Gumin's algorithm functions using two distinct models, an overlapping model or a tiled model [8]. The difference between the two lies mostly on the type of information that the user provides as input and on the type of output that the algorithm will produce. Specifically, the overlapping model takes as input bitmaps and produces bitmaps that are locally similar to the input, while the tiled model takes as input a set of tiles and a set of adjacency rules and generates a tilemap. Apart from that thought, there are some differences in the way the algorithm functions when using these models. In the case of the overlapping model we talked about local similarity. This means that the NxN patterns that can be observed in the input bitmap should also appear in the output image and that the distribution of these patterns is similar in both images. In other words, both the input and the output bitmap are images that showcase similar visual features and characteristics, which appear with the same frequency on both of them. This is done by initializing the output bitmap in a completely unobserved state, meaning that each pixel of the image can take any possible color value that has been observed in the input image. Then the core of the algorithm is a loop of observations and propagations. On the observation step, an NxN region of the unobserved image is selected according to its Shannon entropy [2]. The entropy is calculated as the sum of the frequency that the valid patterns for one pixel appear in the input bitmap and is the value that ensures that the patterns are equally distributed on both images. A state is then randomly selected for this region from all the possible states and the state of this region collapses. Then, on the propagation step the information propagates through the whole output images and the number of possible states for each pixel decreases accordingly. Once all the pixels have been assigned a state, the output image transitions to a completely observed state and the wave function collapses. However, there is a chance that during the propagation step a certain pixel of the output image might"run out" of possible states, meaning that there is no value that can be given to it which will lead into

an acceptable result. This is what Gumin calls a contradiction and thus the algorithm can not proceed any further and no output can be produced. In this case, the algorithm has to back-propagate and redo all the steps that have been done so far.

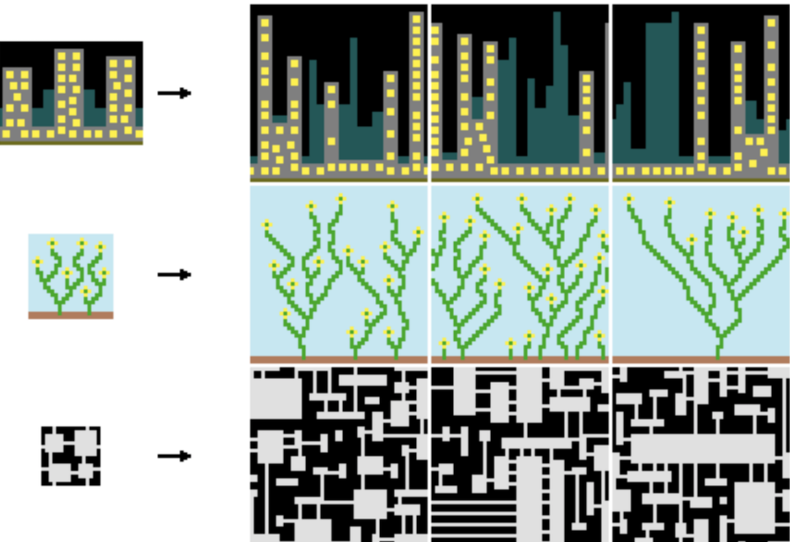


Figure 2.2: Examples of various bitmap inputs and their outputs using the overlapping model WFC. Source: Maxim Gumin's github [8]

A step-by-step presentation of the WFC algorithm using an overlapping model can be seen below:

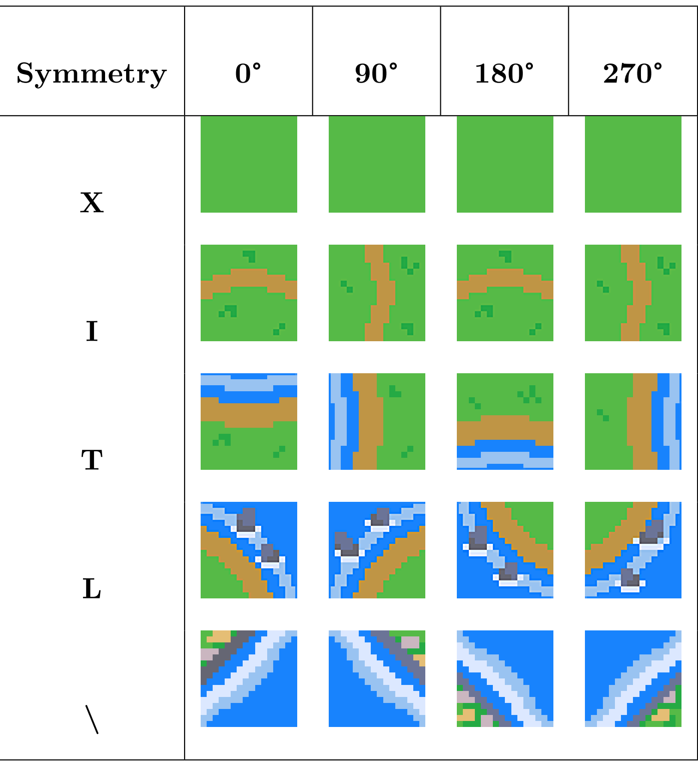
1. Find all the NxN patterns on the input bitmap
2. Represent the output image as an array with the dimensions that are provided by the user. Each element of the array represents a state of an NxN region of the output. This state is a superposition of the NxN that the algorithm recognized on step 1 with boolean coefficients, false meaning that this pattern can not be used in that specific region and true meaning that it has not yet been forbidden and might be the final state of that region. This array is basically the wave.
3. Initialize the array in a completely unobserved state. This essentially means that all the elements of the array have a true boolean coefficient.
4. Initiate the observation-propagation loop circle:
   (a) Observation:
      i. Find the element of the array that has the lowest non zero entropy. If no such element exists move to step 5.
      ii. Assign a definite state to the chosen element according to its coefficients and by taking into account the distribution of the NxN patterns in the input.
   (b) Propagation: propagate the information across the rest of the array (wave), reducing the number of true boolean coefficients each element of the array has.
5. When step 4 is completed and the wave has collapsed two outcomes can occur. Either all the elements are in a completely observed state, so we got a valid output, or there a contradiction took place and we got no output at all.

2.2.2 Tiled Model

In the case of the tiled model the core idea is the same, however the input and the constraint solving function differs. To be more specific, in the tiled model, or simple tiled model as Gumin calls it, a set of tiles is used as input instead of a bitmap image. A tile is basically, a small image of NxN pixels that is used in conjunction with other tiles in order to make a larger image. The constraint rules in this case are translated to adjacency rules. By adjacency rules we mean that each of the tiles is accompanied by adjacency data that define which tile can be placed next to it. However, lists of all the possible rules that define the adjacency of the tiles can be very long, especially when there are a lot of available tiles. To prevent that Gumin uses a symmetry system which is used to generate the above adjacency rules, by simply assigning a symmetry type to each tile and defining one adjacency rule for each pair. The symmetry types used in Wave Function Collapse are five and all of them have an assigned letter to them, from "X", "T", "I", "L" and "\". The symmetry of the tile is similar to that of the letter that is assigned to its symmetry type and defines the number of unique tiles that each tile can produce when rotated by 90 degrees [32]. On Table 2.1, all the unique rotations for each of the symmetry types can be observed. For example a tile with symmetry "X" in reality is one unique tile not matter how many times we rotate it by 90 degrees, while a tile with symmetry "T" can be translated into four tiles which are created every time we rotate the initial tile by 90 degrees.

Figure 2.3: An example tileset and the tilemap generated via the Simple Tiled WFC model. Source: Maxim Gumin's github [8]

The output image when using the tiled model is initialized as a tile map where all the slots can be any of the given tiles. Entropy in this case is measured as the total sum of tile weights per slot [32]. The weight of each tile is defined by the user and it defines the frequency of its appearance in the output image. The higher the weight of a tile the more frequently this specific tile will appear on the output image. So, in this case, on the observation step the slot of the initialized tile map (which serves as the wave) with the lowest entropy is chosen. At the start, all slots have the same entropy, since all slots can take any of the given tiles, so the algorithm picks one at random. This randomness is determined by a random seed that can either be provided by the user or get generated automatically through a random generator. These random seeds dictate the random start of the algorithm and have a huge impact on the outcome, which is the reason behind WFC's procedural generation potential. Once a slot is selected, a random tile from all the possible



Symmetry	0°	90°	180°	270°
X				
I				
T				
L				
\				

Table 2.1: The rotational changes of the tiles according to their symmetry type.

tiles that this slot can take is selected. This random tile is also related to the random seed. Then on the propagation step, this information is spread across the whole tile map based on the adjacency rules given. Basically, that means that some of the tiles become unavailable for the neighboring slots of the chosen slot, because according to the rules, some tiles can not be adjacent to each other. If during the propagation step one of the neighboring slots "run out" of tiles that can be assigned to them, then we have our contradiction condition and an output cannot be produced.

A more detailed description of the algorithm that takes place on the simple tiled model is shown below:

1. Create the output image as an array with the same dimensions. This array is the wave. Each element of the array represents a part of the output image that has the same dimensions as those of the tile and contains a list of possible modules/tiles, which is the possibility space of this element.
2. Initialize the array in a completely unobserved state. This essentially means that all the elements of the array can be any of the possible modules in the possibility space.
3. Initiate the observation-propagation loop circle:
   (a) Observation:
      i. Find the element of the array that has the smallest possibility space. Essentially, this element is the one which has the shortest list of possible tiles. If all the elements have one tile or no tiles left in their possibility space, break the circle and go to step 4.
      ii. Randomly select one of the tiles from the possibility space of the chosen

element and delete the rest of the tiles. This element is then considered collapsed.

(b) Propagation: propagate the information across the rest of the wave, reducing the number of available tiles for the adjacent to the chosen slot slots, based on the adjacency rules that derive for the symmetry of the tiles and the simple left-right rules given by the user.

4. When step 3 is completed and the wave has collapsed two outcomes can occur. Either all elements contain zero tiles, in which case a contradiction occurred and there was no output, or all the elements of the wave contain one module and the output image was created.

2.2.3 3D Wave Function Collapse

In the above section we mainly talked about the generation of two dimensional content, whether that was a bitmap or a tile map. The Wave Function Collapse algorithm, thought, can also work with 3D content. The main model that is used in this case is the overlapping model, since there are no tiles. Briefly, 3D Wave Function Collapse takes as input a 3D sample of the content that has to be generated. Then the user has to define the dimensions of the patterns that have to be recognized. However, when compared to the 2D equivalent this time the patterns are blocks rather than 2D pixel formations. In other words, the algorithm has to create a 3D model using the pattern blocks that are derived from the sample. Nonetheless, the functionality of the algorithm is the same as the one we discussed in section “Overlapping Model” (2.2.1).

2.2.4 Wave Function Collapse Applications

As mentioned earlier, from the moment Maxim Gumin released the Wave Function Collapse algorithm, many developers and researchers started experimenting with it and spreading the word across social media (especially Twitter). In this section, we are gonna present some of the most remarkable ones.

Starting off, WFC is used on the real-time tactics game “Bad North” (Plausible Concept, 2018). Specifically, Oskar Stalberg, one of the developers of the game, explored the 3D content creation capabilities of the algorithm and used it to generate procedural spheres with custom tilesets, generate buildings which can be infinitely extended on the Y axis as well as generating islands. However, Stalberg was not the only one to experiment with 3D content generation using WFC. As a matter of fact, Marian Kleineberg [16] managed to implement an application on Unity3D that can generate an infinite 3D world. Briefly, what Marian did was give as input to the algorithm around 100 unique 3D blocks and created custom constraints for each of them. Then, by implementing some minor modifications to the main algorithm, they managed to generate a fully traversable 3D city that can extend infinitely towards any direction. Two more games that utilize the WFC algorithm are “Caves of Qud” (Freehold Games, 2015), a science fantasy roguelike, and “Townscaper” (Raw Fury, 2020), an instant town building game. All three of the aforementioned games use design focused variations of WFC in order to procedurally generate their content. Apart from games though there has been some academic research on how to manipulate WFC for more game design focused content generation. Specifically, in their research Arunpreet Sandhu et al. [32] are discussing how the WFC algorithm can be enhanced by adding level design constraint, such as non-local constraints in the form of objectives on the map or

special weights for the patterns or the tiles used in the algorithm, while Tobias Moller et al. [26] used a pre-propagation technique that would ensure that there are traversable paths that would in turn guarantee that the generated level can be finished.

Furthermore, an application that is really close to what this approach is about is that of Krolikowski et al. [17]. This application combines Picbreeder [33], an online collaborative tool that can evolve images based on the user's preferences, with the WFC algorithm in order to generate Zentangles [11]. In their approach, Krolikowski attempted to create an automated evolution as well and although it managed to have some success on it, the results that came from the collaboration with the user were more pleasing towards them.

Last but not least, a very interesting application is that of Joseph Parker [29] (also known as Selfsame on the platform of itch.io). Parker implemented an application on Unity3D for generating content using both the models of the WFC algorithm. Briefly, the user of the application has to "draw" a sample image using a tileset of her choice. This tileset is the same tileset that will be used as an input for the Tiled Model of the WFC algorithm and the only thing that has to be specified by the user is the symmetry type of each of the tiles. Once the input sample image is complete, if the user has chosen the Overlapping Model the application identifies the patterns of the drawn image and generates the appropriate results using the WFC algorithm. If the user has chosen the Simple Tiled model the application compiles the drawn image and creates the ruleset that will be used to create the output image. What is interesting here is the way the rulesets are being generated. Usually, a ruleset is provided as input by the use of the WFC algorithm and the user herself has to manually create it. In Parker's approach, though, the ruleset is being generated automatically based on the sample input image that the user has to draw on the application's canvas, much like how the Overlapping WFC model works. The way Parker implemented this is by making an algorithm that scans the input image and determines the adjacency rules according to the positional relationships of the tiles on this image. However, he does it in a more clever way, which takes into account the rotated versions of the tiles as well. This means that a pair of tiles can produce more that one rule based on how many rotated versions of the same tile exist according to the symmetry system of the WFC algorithm. Apart from the ruleset, the algorithm can also identify which tiles are being used from the tileset, in case the image drawn by the user does not contain all of them. This can help make the propagation and eventually the generation of the output image faster, since the available tiles that can be used for each slot of the output image can be significantly reduced this way. Once the drawn image is compiled, the application proceeds to generate the output image using the WFC algorithm's tiled model.

A more thorough view of the way Joseph Parker generated the tileset based on the sample input image can be seen below:

1. Translate the sample input image into a two dimensional array of tiles. The positions of the elements in the array correspond to the position of the tiles in the input image.
2. Start from the element of the array at position (0,0).
3. Go through all its neighbors. The neighbors are the Von Neumann neighbors (right, left, up and down).
   (a) Create the pairs of tiles according to the positions of the tiles, starting by creating the left to right rule, with the left tile being the element that is being scanned at the time.
   (b) Continue by generating the rest of the pairs. This is done by rotating both the tiles of the pair in order to change from a right to left rule to a left to right rule

for example.

4. If the generated rules do not already exist in the ruleset, add them. Otherwise, continue to step 5. If the element is the last element of the array go to step 6.
5. Continue to the next element of the array.
6. Finish when all the elements of the array have been scanned.

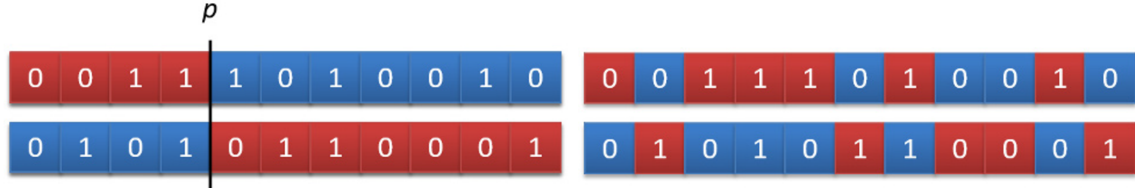
2.3 Evolutionary Algorithms

The idea of Evolutionary Algorithms derived from the Darwinian theory on evolution by natural selection [42]. Much like what Darwin proposed about the progression of humanity through the survival of the fittest and their reproduction to produce even fittest individuals, evolutionary algorithms' core is a population of solutions that are kept in memory at any given time. Then the general idea of the algorithm is to optimize the "reproduction" of the individuals, by generating many solutions, keeping the good ones and throwing away the bad ones and then generating new solutions based on the good ones. This is why evolutionary algorithms are global optimization algorithms, since they search many points in the search space simultaneously.

The main components of an evolutionary algorithm are its population, its fitness function, its selection method, its reproduction method and, finally, its replacement strategy. The population is basically the N number of solutions at any given time, while its fitness function is the function that evaluates each solution and assigns a fitness value to them. The higher the fitness of a solution the better this solution is.

When it comes to selection there are three techniques that are most commonly used: proportional to fitness, proportional to rank and tournament selection [9]. The first technique usually uses a method called roulette wheel to select the parents. Briefly, a proportional value is attributed to each individual based on their fitness. The fitter individuals have a higher value and thus a higher probability of being selected. Considering that the wheel is the probability space of all the individuals, a random point on it is chosen and the individual that corresponds to that point is the one that is being selected. The second technique is also known as ranking. In this case, the individuals are not chosen based on their fitness but based on their rank. The rank of an individual is given to it according to its fitness. When a rank is selected every individual has the same chance to get chosen, which makes the selection of low fitness individuals more likely to happen. Finally, the tournament selection technique randomly selects a subset of individuals from the population and runs a "tournament" between them. At the end of it, the individual with the overall highest fitness is the one that is chosen. Although, with this technique there is a higher chance of selecting a high fitness individual, the lower fitness ones are not excluded.

There are two main reproduction methods when it comes to evolutionary computation: crossover and mutation [42]. Crossover, also known as recombination, is the equivalent of sexual reproduction in the natural world. It basically refers to the creation of an offspring through the combination of two individuals/solutions, who are usually called parents. The idea is that if the two parents are chosen out of the population as good solutions, then a solution that is generated, the offspring, by combining those two individuals ought to be good as well or even better. In the case of mutation, the idea is that sometimes if a good individual is slightly changed then the new offspring will most likely be good too or even better than the parent. Offspring operators can vary depending on the representation of the solutions. For example, if the solutions are represented as strings or vectors, operators



(a) Illustration of a 1-point crossover. (b) Illustration of a uniform mutation.

Figure 2.4: Two examples of applying crossover on a binary chromosome. Red and blue genes represent the two different offspring emerged from each crossover operator. Source: Game AI Book [42]

(a) Illustration of a bit flip mutation. (b) Illustration of an inversion mutation.

Figure 2.5: Two examples of mutating a binary chromosome. In each picture the top chromosome is the selected one and the bottom chromosome is the mutated one. Source: Game AI Book [42]

such as uniform crossover, where a coin is flipped to randomly select values from each parent for each position of the offspring, or one-point crossover, where a random position in the offspring is chosen and its values are the values of parent 1 before the chosen point and the values of parent 2 after it, can be used. Illustrations for both of these operators can be found on Figure 2.4. In a similar manner mutation operators can either be bit-flip mutation, where a number of positions are selected on the parent and the values on these positions are changed to any other random possible value, or an inversion mutation, where two positions are randomly chosen on the parent and all the values that are between these two positions are changed to another possible value. Figure 2.5 showcases examples of both these methods.

Finally, some of the most popular replacement strategies when it comes to evolutionary algorithms are the generational, the steady state and the elitism [42]. According to the generational strategy the offsprings that are created replace all of their parents even if their fitness is worse than their parents. On the contrary, the steady state strategy suggests that one offspring is created at a time and it replaces the worst parent if and only if it has a higher fitness than that specific parent. The elitism strategy is somewhere in the middle. In this strategy, the offsprings that are created replace their parents, but a certain percentage of the best parents survive.

The basic template for an evolutionary algorithm, also known as a generation, is as follows:

1. *Initialization*: The initial population of N solutions is created randomly, which means that each individual is a random point in search space. Sometimes the initial population is created by already existing good individuals.

2. *Evaluation*: In this step a fitness function is used to evaluate all the individuals of the

population and assign their fitness values.

3. *Parent selection*: The individuals that are gonna be used for the next reproduction step are selected using one of the techniques that were discussed earlier.

4. *Reproduction*: Offsprings are produced by either crossovering the selected parents or by mutating them or sometimes a combination of the two.

5. *Replacement*: In this step, the parents and/or the offsprings that will make it to the next generation are selected, using one of the strategies that were discussed above.

6. *Termination*: The evolutionary loop is terminated if the maximum number of generations or evaluations has elapsed (exhaustion) or the highest fitness has been attained by any individual (success) or if any other termination condition is met.

7. If the loop does not terminate, go back to step 2.

Evolutionary algorithms can also be categorised in specific families according to their objective or their evolution strategy. One of the most popular families of evolutionary algorithms is genetic algorithms [43] (GAs). Their main characteristics are their reliance on crossover rather than mutation for variation , fitness-proportional selection and solutions being often represented as bit-strings or other discrete strings. Another variation of evolutionary algorithms emerges if the solutions must not only be the fit, but also be feasible. This can be translated into a combination of evolutionary algorithms and constraint solving problems. However, when evolutionary algorithms are used for constrained optimization, a new challenge appears, which derives from the fact that the evolutionary operators of mutation or crossover cannot guarantee that the solution produced will also be feasible. Some approaches to this problem are repair, which can be any attempt to turn an infeasible result into a feasible one, and genetic operators modifications so that the probability of producing infeasible solutions is smaller. The most popular approach, though, is that of the feasible-infeasible 2-population algorithm [14] which works on evolving two populations at the same time, one that has only feasible solutions and one that has only infeasible ones. Lastly, another popular family is that of multiobjective evolutionary algorithms [39], which considers at least two objective functions, usually conflicting to each other, and searches for a Pareto front of these objectives.

Chapter 3

Methodology

In this chapter the tools that were utilized as they were, like for example the WFC algorithm, and the tools that were implemented, in order to approach our research question, will be discussed. On the first part, the parts that are specific to the Wave Function Collapse algorithm will be presented, while on the second part the evolutionary algorithm implementation and specifically, how the problem was translated into a search space able to be evolved will be presented. Lastly, information about how the output images generated by the WFC algorithm were evaluated and how this evaluation was incorporated into the main evolutionary core of our project will be provided.

3.1 Wave Function Collapse Components

The core functionality of the WFC algorithm is going to be used as a black box and this approach will focus only on the input components of it as well as the output images. The WFC model that was chosen for this implementation is the Simple Tiled one, with some added features from the Overlapping. As it was discussed in Section 2.2.2 the main inputs of the Simple Tiled model are the tileset and the ruleset that accompanies it. In this Section, these two components and how they were implemented for this thesis are going to be presented.

3.1.1 Tileset

In this implementation the focus is generating maps that can be used on 2D adventure or role playing games. Thus, the tileset that was utilized is a generic overworld tileset [1] that can be used to generate landscapes that are most commonly seen in such games, like for example valleys, villages, forests or coast lines. Originally, the tileset contains 1003 tiles [2] (including rotated tiles) and is divided into three types of landscapes: green land, icy land and desert. In this thesis, only the tiles that correspond to the green land were used, which take up a third of the whole tileset for a total of 334 tiles (including rotated versions of the same tile). The final number of tiles was further decreased, because of some issues assigning the correct symmetry type to some of the tiles. For reasons of experimenting and convenience the final group of tiles was divided into 3 separate tilesets: a simple tileset, a medium tileset and a full tileset.

[1] Source of the tileset: `https://mattwalkden.itch.io/rpg-overworld-tileset`

[2] The tileset that we used is the compressed no-animation one that is provided by the creator.

Tileset	Simple	Medium	Full
Number of Unique Tiles	5	11	70
Number of X symmetry type tiles	2	5	35
Number of L symmetry type tiles	2	4	12
Number of T symmetry type tiles	1	2	16
Number of I symmetry type tiles	0	0	7
Number of \ symmetry type tiles	0	0	0
Number of final tiles	14	29	161

Table 3.1: The number of tiles for each tileset.

Each of these tilesets differ in the number of tiles that are part of it and are overlapping in a simplicity ranking manner, meaning that the medium tileset contains all the tiles that are on the simple tileset and the full tileset has all the tiles that appear on the medium tileset, but each time the tileset becomes more complex. On table 3.1 the number of the tiles for each tileset are showcased. The final number of tiles takes into account the tiles that are produced if we rotate the unique tiles. However, as already discussed on section 2.2.2 not all types of tiles produce new tiles when they are rotated. Specifically, as presented on table 2.1 the tiles of type "X" have only one rotated version which aligns with the non-rotated/unique version of the same tile, while the tiles of type "L", "T" and "\" have 4 different versions of the same tile according to its rotation and tiles of type "I" have two versions. All the tiles that will be used in this thesis, regardless of tileset, are individual $48x48$ pixels in size images.

Lastly, in order for the WFC algorithm to produce the correct results, each of the unique tiles (the original not rotated tile) has to be assigned with its symmetry type. For this thesis, we did this part manually, which means we went through all of the tiles that we used for each of the tilesets and assigned its respective symmetry type. Apart from the symmetry type each of the tiles has information about its walkability (whether a player would be able to pass through it or not), its source image file location, which is required for the WFC algorithm in order to generate the output image and its land state, for example whether this tiles corresponds to a grass tile or a sea tile.

3.1.2 Ruleset

The tileset on its own is not enough for the Wave Function Collapse to produce an output. A ruleset that will establish the way the tiles can be placed together is also required for the algorithm to function properly. As it was discussed in section 2.2.2, each ruleset consists of single rules that are simple left and right adjacency rules, also called neighboring rules because they establish left neighbor to right neighbor relationships. In this thesis, Joseph Parker's [29] approach on generating the ruleset necessary for the WFC algorithm through a

sample input image will be utilized to automatically generate the rulesets. Specifically, the rulesets will not be crafted manually as it is suggested by the Simple Tiled Model of WFC, but instead sample maps will be created using the three groups of tilesets that we described above and will be used in combination with Parker's algorithm to create the ruleset that correspond to them.

3.1.3 Wave Function Collapse Setup Summary

In conclusion, the tileset is split into three sub-tilesets of different complexities, which are going to be used to conduct distinct experiments. In these experiments, sample input images will be generated and Joseph Parker's approach will be utilized to create the rulesets necessary for the WFC algorithm. The part of the WFC algorithm that is associated with the procedural generation of the images will be used as a black box, meaning that none of its internal parameters will be changed. Finally, the output images generated by the algorithm will be saved in both image form and array form, in order to be made accessible for further evaluation.

3.2 Evolutionary Algorithm

The next step in this approach is to implement our evolutionary algorithm. This in turn means that we have to go through all of the components that an evolutionary algorithm needs in order to function properly, adjust them to our approach and implement them. As discussed in Section 2.3 the main components of an evolutionary algorithm are its population, its selection and replacement method, its evaluation/fitness function and the genetic operators.

3.2.1 Population

The first thing that needs to be defined is the genes that make the population of our evolutionary algorithm. In this case, each gene corresponds to a sample input image which serves as its phenotype. This image is translated into a two-dimensional array of tiles and a list of left-right neighbor rules. These two are part of the genotype of the image, although only the array of tiles is getting evolved. The ruleset is automatically generated and is used as input for the WFC algorithm. Each of the tiles of the array contain some information themselves, which correspond to the name and state of the tile, the source image and finally its bitmap. The source image is used for recreation purposes, while the bitmap will be used on our fitness function later. Furthermore, the genotype contains some info about the random seeds that have already been tested for generating content with WFC, but they are not part of the evolutionary process, at least for the early parts of our implementation.

The population of the evolutionary algorithm is initialized randomly. This means that all the genes of the population start with a phenotype image that contains random tiles picked from the given tileset with equal probability. On Figure 3.1 some examples of how the phenotype images of the randomly generated genes look like for all the three tilesets are presented. Once a new gene is created, the ruleset that corresponds to its image is generated automatically using the algorithm that Joseph Parker [29] implemented on his unity application. Once the population is initialized all the new genes are evaluated using the fitness function we will discuss on Section 3.2.3 and they are sorted accordingly with the fittest gene being first on the list.

Figure 3.1: Three examples for the phenotype images of the randomly initialized population for the simple (left), medium (middle) and full (right) tileset.

3.2.2 Gene Selection Method

The next step after initializing the population is to select the genes that will get mutated or crossovered. In this thesis, we are going to use a roulette selection technique for the selecting of the genes. A roulette wheel selection, as discussed on Section 2.3, assigns a proportional value to every gene according to their fitness, with the fittest gene having a higher probability to be picked. The random point on the probability space is translated into a random point in the cumulative fitness of all the genes and the gene that is selected is the one whose fitness reaches that point, if we add the fitness of all the genes one at a time. We repeat this procedure until the amount of the selected genes is equal to the population size minus the number of genes that we want to survive for the current population. The selected genes can contain the same gene more than once, since we don't immediately remove a gene from the population after it is selected.

3.2.3 Fitness Function

In this thesis we evaluate the genes using a pixel similarity fitness function. When we talk about pixel similarity between the tiles of an image, we refer to whether the pixels at the edges of the tiles that make the image have the same color values as their adjacent tiles or not. The idea behind this is that each of the images that are getting generated along with the genes of the population are basically an array of tiles, which are also smaller scale images placed next to each other to create the bigger image. So, if these individual images (the tiles) have similarly colored edges, they will fit aesthetically when they are put next to each other and thus they will create an image, or a map in our case, that will at least be consistent in terms of aesthetics.

Having that in mind, we calculate the fitness of each gene as the percentage of the same colored pixels, that appear on the edges of the tiles that are adjacent to each other, over all the adjacent pixels that can be calculated on the image. For example, a 2 tiles by 2 tiles image made out of tiles of 10 by 10 pixels has a total of 80 adjacent pixels that can be grouped into 40 pairs. If only 40 of these pixels are the same color with their adjacent pixel then the pixel similarity and thus the fitness would be $0,5$. The tiles that are considered adjacent to a tile are those that are its Von Neumann neighbors, or in other words the four tiles that are up, down, left and right from it. We don't take into account the tiles that are diagonally connected to each tile. An algorithmic representation of how the fitness is calculated can be seen on Algorithm 1, where *image* is the array of tiles that is the

Figure 3.2: Visual representation of pixel similarity calculation. The tile highlighted by the red square is the tile for which we calculate the pixel similarity, while the blue lines indicate which edge pixels are taken into account from the adjacent tiles.

genotype of the gene, $imageWidith$ and $imageHeight$ are the width and height of the image (measured in tiles) and $tileSize$ is the width or the height of the tiles (measured in pixels), which in our case both are 48. Furthermore, a visual representation of a step in the calculation of the fitness of the gene is shown on Figure 3.2.

Once the pixel similarity fitness of the gene is calculated, we enforce a WFC algorithm constraint on it. Specifically, what we do is run the WFC algorithm using the phenotype image of the gene and the ruleset that is generated through it as inputs, along with a randomly chosen seed. If the WFC algorithm does not manage to produce an output image, regardless of its quality (it doesn't have to be a "flawless" image), the fitness of the specific gene is set to zero. The reasoning behind this is that we don't only want our evolutionary algorithm to produce good quality input images, but also be able to actually produce results through the WFC algorithm in general.

The fitness function described above is used for the most part of this thesis. On one of the experiments that we did towards the end of our experimentation phase we use a modified version of this fitness function, which we will discuss further in Section 4.6.

3.2.4 Genetic Operators

In this thesis, the genetic operator that we are going to utilize is mutation. Once all the genes have been selected we proceed with mutating them. The mutation process technically refers to the replacement of one or more tiles on the phenotype image of the gene. More specifically, every time we mutate a gene, we select one or more tiles of its phenotype image and change them into new ones that we pick out of the available tiles of the given tileset. This process is the core of our project and most of the experiments conducted are focused on searching for the optimal way to implement it.

The mutation algorithm that we are going to use can be separated into three steps: the selection, the removal and the replacement steps. Throughout this thesis we are going to use 8 different mutation operators. Each of them features a different selection method for choosing the tile that needed to be changed or a different technique for picking the tile that would be placed on the empty slots of the image or even a different amount of tiles that

Algorithm 1 Pixel Similarity Fitness

```
1: for y = 0 to y = imageHeight do
2:   for x = 0 to x = imageWidth do
3:     if x + 1 < imageWidth then
4:       for k = 0 to k = tileSize do
5:         pixelsCounted = pixelsCounted + 1
6:         if image[x, y].pixel[tilesize − 1, k] = image[x + 1, y].pixel[0, k] then
7:           samePixels = samePixels + 1
8:         end if
9:       end for
10:    end if
11:    if x − 1 ≥ 0 then
12:      for k = 0 to k = tileSize do
13:        pixelsCounted = pixelsCounted + 1
14:        if image[x − 1, y].pixel[tilesize − 1, k] = image[x, y].pixel[0, k] then
15:          samePixels = samePixels + 1
16:        end if
17:      end for
18:    end if
19:    if y + 1 < imageHeight then
20:      for k = 0 to k = tileSize do
21:        pixelsCounted = pixelsCounted + 1
22:        if image[x, y].pixel[k, tileSize − 1] = image[x, y + 1].pixel[k, 0] then
23:          samePixels = samePixels + 1
24:        end if
25:      end for
26:    end if
27:    if y − 1 ≥ imageHeight then
28:      for k = 0 to k = tileSize do
29:        pixelsCounted = pixelsCounted + 1
30:        if image[x, y − 1].pixel[k, tileSize − 1] = image[x, y].pixel[k, 0] then
31:          samePixels = samePixels + 1
32:        end if
33:      end for
34:    end if
35:  end for
36: end for
37: fitness = samePixels/pixelsCounted
```

Genetic Operator	Selection	Removal	Repair
Operator 1	Randomly select a tile of the image	Remove the selected tile from the image	Place a random tile from the tileset at the empty slot
Operator 2	Randomly select a tile of the image	Remove the selected tile from the image	Repair the empty slot deterministically
Operator 3	Roulette selection of one tile	Remove the selected tile from the image	Repair the empty slot deterministically
Operator 4	Roulette selection of two tiles. The second selection does not take into account the first repair.	Remove both tiles from the image one at a time	Place new tiles on both slots deterministically one at a time
Operator 5	Roulette selection of two tiles. The second selection takes into account the first repair	Remove both tiles from the image one at a time	Place new tiles on both slots deterministically one at a time
Operator 6	Roulette selection of one tile	Remove the chosen tile and its Von Neumann neighbors one at a time	Deterministically place a new tile at the empty slots of the image one at a time
Operator 7	Roulette selection of one tile	Remove the tile and its Von Neumann neighbors from the image all at once	Deterministically repair the image starting from the empty slot that has the most non-empty neighbors
Operator 8	Roulette selection of one tile	Remove the tile and its Moore neighbors from the image all at once	Deterministically repair the image starting from the empty slot that has the most non-empty neighbors

Table 3.2: All the genetic operators and their corresponding methods that were used to mutate our genes.

would be removed from the image. On Table 3.2 we showcase all the genetic operators that we implemented in this thesis and the specific selection, removal and repair method that they utilize.

3.2.5 Replacement Method

Once all of the selected tiles have been mutated, the offsprings that were created, replace the least fittest individuals of the population. The replacement method that we used is elitism and specifically, we replace the whole population apart from the 10 fittest parents. We remove the genes that are at the bottom of the population list (since the population is sorted from higher to lower fit individuals) until the population count is 10. Then, we evaluate all the new genes, we add them to the population and we sort the population the same way as before.

Chapter 4

Experiments

In this chapter the experiments that were conducted along with the results that they produced are going to be presented. Briefly, we will first discuss how we will evaluate the performance of our algorithm and the setup that we used for our experiments. Then we will present the experiments that aimed on finding the optimal parameters for our evolutionary algorithm. Finally, we will perform an expressivity analysis on the results that were generated from the WFC algorithm and we will also test the robustness of our algorithm on random tilesets.

4.1 Performance Metrics

The goal of this thesis can be narrowed down to creating good input images that can then be used as input for the WFC algorithm and get good output images. This is reflected in the way the function fitness is implemented, since an image that is created by tiles that feature high pixel similarity values, should theoretically be a good looking image. Furthermore, we want the evolved image to use a wider range of tiles from the given tileset, so that they are not very simple and feature interesting patterns. On Figure 4.1 an example of a simple image accompanied by an example of a more complex one are presented. Although simple images can have a high pixel similarity value, they do not provide any helpful information regarding the playability of the generated maps or the creative capabilities of the algorithm.

The metrics that we used to evaluate the results in this thesis are:

Figure 4.1: Two image of different KL divergence values. On the left we have an image with a KL divergence of 0.53 and on the right one with a KL divergence of 1.81 (same tileset).

- The **final best fitness** of the population. This corresponds to the highest fitness value that we got at the end of the evolutionary algorithm across all the genes that survived until the end.
- The **final average fitness** of the final state of the population. Basically, this is the average fitness value of all the genes that survived at the end of the evolutionary process.
- For the two metrics above we also calculated their **confidence intervals**. In this thesis we are going to use a 95% confidence interval so that we can get an idea of whether the fitness values across all the runs were relatively close to each other or not.
- The percentage of perfect final results we got over all the runs that we did for one experiment, which we call **success rate**. A run has a perfect result if the gene with the final best fitness has a "flawless" image as their phenotype, in the sense that there is no tile on this image that looks out of place.
- The **average time** that the algorithm had to take to complete a full evolutionary circle.
- Last but not least, we want to make sure that the evolved images are not very plain and uninteresting, but rather they are complex and feature a variety of tiles. To do this we use the **Kullback Leibler (KL) divergence** of the image that corresponded to the gene with the final best fitness of each run as the final metric. The KL divergence is the expectation of the logarithmic differences between two probability distributions P and Q defined on the same probability space X and it is calculated as:

$$D_{KL}(P||Q) = \sum_{x \in X} P(x) log\left(\frac{P(x)}{Q(x)}\right) \tag{4.1}$$

The KL divergence can be altered and used on an image to calculate the probability distribution of a tile pattern over all the tile pattern occurrences, giving an estimate of how many distinct patterns appear in the image. If we replace tile patterns with individual tiles then we can get an estimate about how frequently each of the tiles appear in the image. Specifically, following Lucas and Volz's implementation [22], we set $Q(x)$ as the probability of each tile of the tileset to appear on the image. This value is the same for every tile, since all the tiles can appear on the image with the same probability. For example, a tileset that has 5 tiles in it, the Q value for all the tiles would be $1/5$. As for $P(x)$ we use a frequentist approach and we calculate it as the number of times tile x appears on the image divided by the total number of tile occurrences. Furthermore, the probability space X is converted into the tileset that is being used and x can be any single tile of this tileset. The equation that computes $P(x)$ is as follows:

$$P(x) = \frac{C(x) + \epsilon}{(C + \epsilon)(1 + \epsilon)}, \tag{4.2}$$

where ϵ is a small no-zero constant that is added in order to prevent division with zero errors, $C(x)$ is the how many times tile x appears on the image and C is the

sum of $C(x)$ all over $x \in X$. Once we calculate the KL divergence of the image using Equation 4.1 we can estimate how complex the image is by how low the KL value is. Specifically, the lower the value the better, since a low value translates into less frequent appearance of the same tiles on the image, which in turn makes the image more interesting and complex. On Figure 4.1 we can see two images that have different KL values and we can also observe the difference between a low and a high KL divergence image. Although, KL divergence is a good way to tell the complexity of an image, it is very reliant on the size of the tileset and the size of the image, meaning that the larger the tileset gets without the image getting bigger, the higher the KL divergence will get. However, in the scope of this thesis, this was not a major issue since we could still compare the complexity of the images that were generated using the same tileset.

4.2 Experimental Setup

Each of the experiments that we are going to conduct in this thesis were repeated three individual times, one for each of the three tilesets. The tilesets are the same as the ones that we already discussed in Section 3.1.1. For each experiment we run our evolutionary algorithm 40 times/runs and we save the information that we got from each run as well as the average of all the runs. Each of the runs consist of 50 generations, while the population count can differ between experiments. Specifically, for the first group of experiments the number of individuals in the population will be 20, while in the second group we are going to determine the optimal number of genes that we need, which we will use for the rest of the experiments. Lastly, the sample images that are going to be used as the phenotype for each gene will be of a $5x5$ resolution, while the images that the WFC algorithm will produce will be of $20x20$ resolution.

Furthermore, in regards to the tileset that we used for all the experiments, we went through each of the tiles and determined their symmetry type. In this tileset there was no tile that had a symmetry type “\”. However, there were some that have symmetry types of “T”, “L” and “I”. This means that for these tiles we had to create the equivalent images for the rotated versions. For this part of our implementation we did this manually and also we went through some of the tiles and “corrected” some parts of the tiles that were bound to create issues when we had to calculate the pixel similarity of the images.

4.3 Genetic Operators Testing

The first group of experiments is focused on determining which is the optimal genetic operator for mutating the images of the genes. The operators that we tested are the ones featured on Table 3.2 and the testbed was the same for all of them.

Briefly, the operators, as we discussed in Chapter 3, make changes on the phenotype image of the genes by selecting one or more tiles from the image, removing them and filling up the empty spot with different ones. The difference between these operators come from the methods that are being used to implement each of these three steps. Specifically, the operators are:

1. Operator 1: Randomly select a tile of the image and remove it. All tiles have an equal probability to be selected. Repair the empty slot of the image, by placing a randomly

selected tile out of the given tileset. Again all available tiles have an equal chance of being picked.

2. Operator 2: Same selection and removal technique as Operator 1. When repairing the empty slot, go through all the available tiles of the given tileset and pick the one that will result in the highest pixel similarity value for the repaired image.

3. Operator 3: Use a roulette selection technique to select the tile that needs to be removed. This technique favors the selection of tiles that do not fit in the image. Once a tile is selected remove it and repair the image using the same deterministic method as Operator 2.

4. Operator 4: Same selection technique, but this time we choose two tiles from the image. Both tiles are selected together and then one of the tiles is removed and the image is repaired deterministically. Then the same happens for the other selected tile. In this procedure the selection of the second tile does not take into account the change that happens on the image after the first repair.

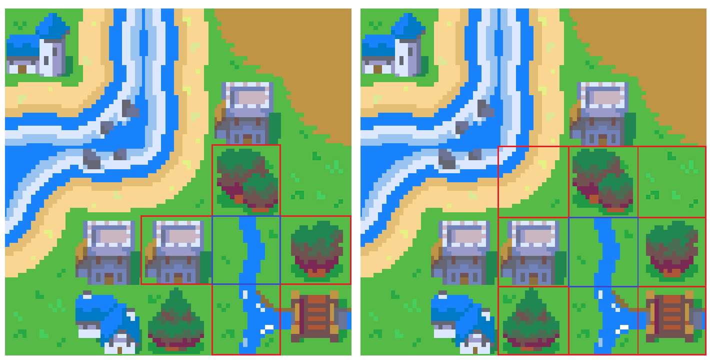
5. Operator 5: Overall the procedure here is the same with Operator 4. The only difference is that this time we select the first tile, remove it and repair the image deterministically and then we select the second tile and repeat the process. This way the second selection takes into account the first repair of the image.

6. Operator 6: Choose the less fitting tile of the image using the same roulette selection technique. Remove it along with its Von Neumann ($r = 1$) neighbors. The removal is done one tile at a time and every time a tile is removed from the image, it is repaired deterministically before the next one is removed. On Figure 4.2 we demonstrate which tiles are being removed.

7. Operator 7: The selection technique is the same roulette technique. Once the tile is chosen, we remove it along with its Von Neumann ($r = 1$) neighbors all at once. Then we repair the empty slots of the image deterministically, starting from the slot that has the most non-empty neighboring slots.

8. Operator 8: Same process as the one described on Operator 7, but this time we remove the Moore neighbors of the chosen tile along with the chosen tile. On Figure 4.2 we present a visual representation of the tiles that are removed.

Each of the above methods was tested on all three of the tilesets using a population of 20 individuals. As we can see on Tables 4.1, 4.2 and 4.3 operators 1 and 2, which are the two operators that are based on a random selection method, did not manage to produce perfect results at all for any of the given tilesets. Operator 3 has an overall better performance than the previous two in terms of fitness and although some of the results that we got throughout the runs were perfect for the simple and medium tileset, there were no perfect results for the full tileset and the success rates were not enough to render this operator reliable. The next two operators, 4 and 5, mutate two tiles of the image every generation. Both of these two performed better than all their predecessors, however operator 4 performed worse compared to operator 5. That was to be expected, because on operator 4 the selection of the second tile on the image does not take into account the change that has already occurred on the image. This way the two mutations that happen on the same image might lead into a result



(a) Von Neum. neighborhood. (b) Moore neighborhood.

Figure 4.2: Visual representation of the Von Neumann (left) and the Moore (right) neighborhood removal techniques. The chosen tile is highlighted in blue, while its neighboring tiles are highlighted in red.

Genetic Operators	Final Best Fitness	Final Average Fitness	Success Rate (%)	KL Divergence	Time (min.)
Operator 1	0.69 (±0.008)	0.65 (±0.006)	0%	0.28 (±0.036)	2.7
Operator 2	0.9 (±0.007)	0.86 (±0.01)	0%	0.22 (±0.042)	2.7
Operator 3	0.95 (±0.007)	0.94 (±0.011)	40%	0.21 (±0.034)	2.4
Operator 4	0.93 (±0.009)	0.86 (±0.015)	35%	0.22 (±0.038)	2.7
Operator 5	0.99 (±0.001)	0.93 (±0.012)	100%	0.3 (±0.06)	2.16
Operator 6	0.99 (±0.001)	0.93 (±0.014)	100%	0.35 (±0.085)	2.55
Operator 7	0.99 (±0.001)	0.97 (±0.019)	100%	0.35 (±0.088)	2.7
Operator 8	1 (±0.001)	0.95 (±0.022)	100%	1.16 (±0.126)	2.34

Table 4.1: Average Values (over 40 runs) for each of the Genetic Operator for the Simple Tileset.(The numbers is brackets are the confidence intervals of the corresponding metrics).

Genetic Operators	Final Best Fitness	Final Average Fitness	Success Rate (%)	KL Divergence	Time (min.)
Operator 1	0.68 (±0.012)	0.6 (±0.015)	0%	0.65 (±0.044)	2.32
Operator 2	0.87 (±0.015)	0.79 (±0.022)	0%	0.71 (±0.057)	2.7
Operator 3	0.93 (±0.014)	0.83 (±0.031)	5%	0.67 (±0.059)	2.34
Operator 4	0.9 (±0.02)	0.73 (±0.053)	15%	0.7 (±0.069)	2.1
Operator 5	0.99 (±0.003)	0.81 (±0.048)	85%	1.15 (±0.075)	1.84
Operator 6	1 (±0.001)	0.82 (±0.054)	100%	0.78 (±0.047)	2
Operator 7	0.99 (±0.003)	0.93 (±0.023)	95%	0.83 (±0.072)	2.49
Operator 8	1 (±0.001)	0.91 (±0.018)	100%	1.09 (±0.128)	2.07

Table 4.2: Average Values (over 40 runs) for each of the Genetic Operator for the Medium Tileset. (The numbers is brackets are the confidence intervals of the corresponding metrics).

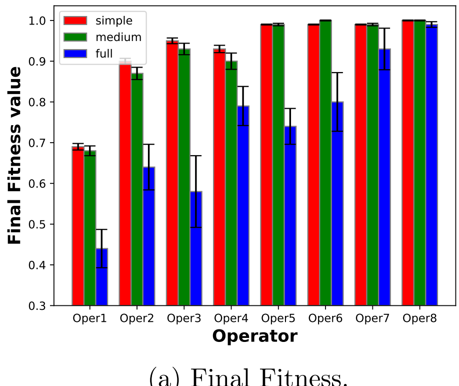
Genetic Operators	Final Best Fitness	Final Average Fitness	Success Rate (%)	KL Divergence	Time (min.)
Operator 1	0.44 (±0.047)	0.29 (±0.037)	0%	1.32 (±0.036)	2.44
Operator 2	0.64 (±0.056)	0.43 (±0.053)	0%	1.44 (±0.045)	2.7
Operator 3	0.58 (±0.088)	0.39 (±0.068)	0%	1.39 (±0.04)	2.52
Operator 4	0.79 (±0.048)	0.51 (±0.044)	0%	1.46 (±0.037)	2.77
Operator 5	0.74 (±0.044)	0.44 (±0.038)	0%	1.6 (±0.061)	2.32
Operator 6	0.8 (±0.072)	0.5 (±0.058)	15%	1.57 (±0.047)	3.45
Operator 7	0.93 (±0.051)	0.69 (±0.058)	75%	1.72 (±0.084)	2.91
Operator 8	0.99 (±0.007)	0.77 (±0.034)	90%	1.97 (±0.121)	2.83

Table 4.3: Average Values (over 40 runs) for each of the Genetic Operator for the Full Tileset. (The numbers is brackets are the confidence intervals of the corresponding metrics).

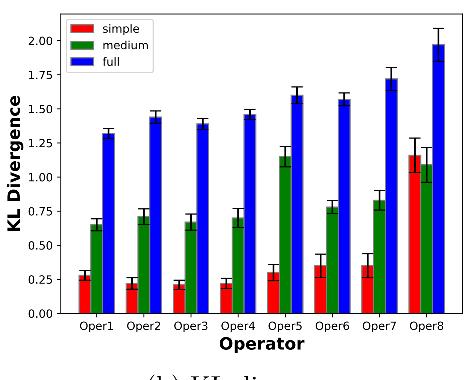
that is not repaired optimally and thus there will be no increase on the gene's fitness. This fact is confirmed by the inability of operator 4 to produce a sufficient number of perfect results, especially on the medium tileset where it has a 15% success rate compared to the 85% success rate of operator 5. Operator 5 on the other hand, is the first operator to achieve 100% success rate on the simple tileset and reached a final fitness average value of 0, 99. It also achieved similar scores for the medium tileset, reaching a final fitness average value of 0, 99 again and a 85% success rate, but failed to do so for the full tileset, on which it produced no perfect results.

The last three operators are the ones that produced the best results in terms of fitness as shown on Figure 4.3, and they are the only ones that managed to produce perfect results on the full tileset as shown on Table 4.3. The results that we got using these operators on the simple tileset are very similar to each other. The only thing that stands out is the KL divergence value of operator 8, which is significantly higher than the rest of the operators. This means that although operator 8 achieved high fitness values, the images that it produced were not very complex or interesting and were very similar to the high KL image that is shown on Figure 4.1. On the medium tileset, the results for all three of the last operators are really close too, with operator 8 still having a higher KL divergence value than the rest and operator 7 not managing to have a 100% success rate. Lastly, on the full tileset we can notice some significant differences between them, with operator 8 outperforming the other two making it the best option when it comes to generating images using a large number of tiles. Operator 6 is the weakest of the three, since only on 15% of the runs it managed to produce a perfect image result, while scoring the lowest fitness values and taking slightly longer to run a full run. Finally, operator 7 results, in terms of fitness and time, were really close to those of operator 8. However, its success rate is 75% which makes it slightly less reliable when it comes to generating good results.

To sum up, Operators 1 to 5 did not manage to generate any perfect image when we used the full tileset, which does not make them good options, considering the goal of this thesis is to create a universal tool that designers can use on any tileset. While operator 6 managed to reach high fitness values and success rates on the simple and medium tilesets, it failed to do so on the full tileset making it only a viable option on small tilesets. Operator 8 managed to have high fitness values across all runs and all tilesets, making it the best option in terms of generating flawless images. However, the KL divergence values of the final images that were produced using this operator are also very high compared to the rest of the operators, meaning that although the images that were created are flawless in terms of pixel



(a) Final Fitness.



(b) KL divergence.

Figure 4.3: The average values (across 40 runs) for the final fitness and the KL divergence for each of the operators on all three tilesets.

similarity, they do not feature a lot of complexity, especially if the tileset is small. This leas to operator 7 being the final choice for this approach. Although operator 7 did not achieve a 100% success rate on the medium and full tilesets, its final and average fitness values were almost as high as the ones of operator 8. Additionally, as we seen on figure 4.3 the images that were produced using operator 7 have significantly lower KL divergence values, meaning that they are much more interesting and complex than those generated using operator 8. Operator 7 produced good results both in terms of similarity and complexity and that is why for the rest of this approach it will be the genetic operator for mutating the genes of the evolutionary algorithm.

4.4 Population Count Testing

Although, genetic operator 7 had a very good performance overall, the fact that on the full and medium tileset it did not manage to have a success rate of 100%, does not make it the perfect candidate when it comes to reliability. That is why the next group of experiments is going to focus on testing whether changing one of the parameters of the evolutionary algorithm, other than the genetic operator, can improve its overall performance. Specifically, in this group we will change the number of individuals of the population and check whether it has a positive or negative impact on the results that are being produced.

For each of the three tilesets, we run a full experiment (40 runs) using a population of 20, 30, 40, 50, 75 and 100 individuals. The rest of the parameters were the same as previously and the genes were mutated using genetic operator 7 only. Tables 4.4, 4.5 and 4.6 present a summary of the results we got for this group of experiments. As we can see, an increase on the size of the population can lead into an increase on the fitness values. Specifically, on Figure 4.4 it is showcased that the final fitness value of the evolutionary algorithm increases when the population count increases regardless of the size and complexity of the tileset. However, some of the experiments in this group still did not manage to achieve a 100% success rate which was our primary goal in this part. In particular, only population sizes greater than 40 managed to have a perfect result on every run. Apart from that we can also see on Figure 4.4 that for population sizes of 50, 75 and 100 there isn't any significant increase in the final fitness, meaning that for all these sizes the results that we got were

Population Count	Final Best Fitness	Final Average Fitness	Success Rate (%)	KL Divergence	Time (min.)
20 indiv.	0.991 (±0.001)	0.975 (±0.019)	100%	0.35 (±0.088)	2.7
30 indiv.	0.992 (±0.001)	0.974 (±0.011)	100%	0.409 (±0.087)	5.1
40 indiv.	0.993 (±0.001)	0.972 (±0.011)	100%	0.452 (±0.099)	8.85
50 indiv.	0.993 (±0.001)	0.975 (±0.009)	100%	0.435 (±0.1)	9.3
75 indiv.	0.994 (±0.001)	0.954 (±0.018)	100%	0.457 (±0.095)	16.5
100 indiv.	0.995 (±0.001)	0.969 (±0.01)	100%	0.505 (±0.088)	23.85

Table 4.4: Average Values (over 40 runs) for each population size for the Simple Tileset. (The numbers is brackets are the confidence intervals of the corresponding metrics).

Population Count	Final Best Fitness	Final Average Fitness	Success Rate (%)	KL Divergence	Time (min.)
20 indiv.	0.993 (±0.003)	0.933 (±0.023)	95%	0.832 (±0.072)	2.49
30 indiv.	0.996 (±0.002)	0.915 (±0.032)	97.5%	0.839 (±0.077)	5.25
40 indiv.	0.998 (±0.001)	0.925 (±0.022)	100%	0.844 (±0.05)	8.7
50 indiv.	0.999 (±0.001)	0.942 (±0.018)	100%	0.825 (±0.059)	11.55
75 indiv.	0.999 (±0.001)	0.949 (±0.011)	100%	0.823 (±0.054)	18.6
100 indiv.	1 (±0)	0.936 (±0.016)	100%	0.881 (±0.051)	21.15

Table 4.5: Average Values (over 40 runs) for each population size for the Medium Tileset. (The numbers is brackets are the confidence intervals of the corresponding metrics).

equally good. Although the increase of the population size above 50 did not have an impact on the final fitness, it did affect the time that the experiment took to complete and as we can see on Tables 4.4, 4.5 and 4.6, with higher population sizes taking significantly longer to complete a single run. Another interesting observation is that for each of the tilesets all population sizes feature similar KL divergence values, which in turn means that the population size that we use does not affect the complexity of the produced images. Based on all of the above observations, for the rest of the experimentation phase we chose to use a population size of 50, because going for larger sizes both did not contribute to having better results and it required more computational time to complete a single run.

Population Count	Final Best Fitness	Final Average Fitness	Success Rate (%)	KL Divergence	Time (min.)
20 indiv.	0.934 (±0.051)	0.685 (±0.058)	75%	1.724 (±0.084)	2.91
30 indiv.	0.99 (±0.01)	0.741 (±0.039)	90%	1.79 (±0.076)	6.9
40 indiv.	0.994 (±0.003)	0.766 (±0.027)	92.5%	1.72 (±0.073)	9.75
50 indiv.	0.997 (±0.001)	0.714 (±0.031)	100%	1.685 (±0.075)	13.5
75 indiv.	0.998 (±0.001)	0.783 (±0.038)	100%	1.685 (±0.069)	21
100 indiv.	0.998 (±0.001)	0.757 (±0.039)	97.5%	1.764 (±0.054)	30.9

Table 4.6: Average Values (over 40 runs) for each population size for the Full Tileset. (The numbers is brackets are the confidence intervals of the corresponding metrics).

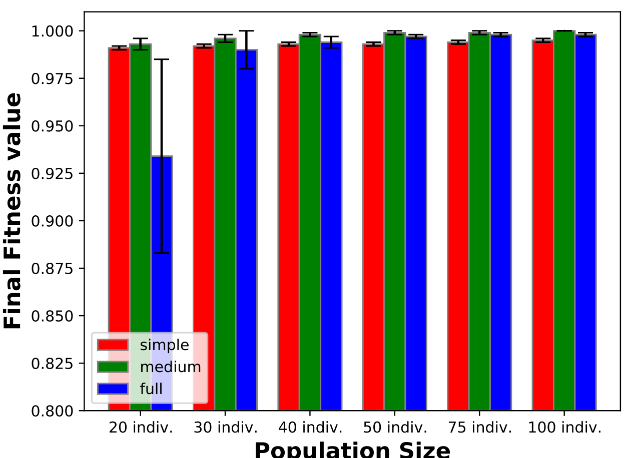


Figure 4.4: Final fitness average values (across 40 runs) for each of the population sizes on all three tilesets.

4.5 Resolution Testing

For the next group of the experimentation we did not change any parameters that are directly connected to the evolutionary algorithm, but instead we wanted to test if the algorithm can still produce good results when we increase the resolution of the phenotype images of the genes. At this point we should mention that when we talk about the resolution of the images, we refer to the number of tiles used to create these images. For example, a $5x5$ resolution image is technically an image that is generated using 25 tiles in total (5 tiles width and 5 tiles height). What we are looking for here is to test whether increasing the resolution will lead into more complex and interesting images, which in turn might affect the complexity of the images and thus the maps that are generated by the WFC algorithm.

In this group the algorithm was tested using 4 different resolutions including the $5x5$ resolution that was already tested on the previous experiment. The new resolutions were $7x7$, $9x9$ and $11x11$ tiles. Tables 4.7, 4.8 and 4.9 present the summary of the results that were produced from all the tests conducted on each tileset. With a quick look we can see that the algorithm's performance in terms of fitness is dropping the higher the resolution of the input images gets. A clearer representation of this can also be seen on Figure 4.5 where the final fitness values that the algorithm achieved are lower for the higher resolutions. This means that the evolutionary algorithm is struggling to create images of higher resolution with the current setup and it could be fixed by slightly tweaking some of the evolutionary parts, like increasing the number of generations or even increasing the population size even more. The decline of the algorithm's performance can also be observed through the success rate. Specifically, the algorithm did not manage to generate perfect results on every run for resolutions higher than $5x5$ on the medium and full tileset, which leads to the assumption that the algorithm will not be able to produce good high resolution images when using a large tileset.

Furthermore, the higher resolution images that are produced have lower KL divergence values. This is to be expected, since the higher the resolution the higher the number of tiles used for a single image. This naturally leads to a higher number of unique tiles used

Resolution	Final Best Fitness	Final Average Fitness	Success Rate (%)	KL Divergence	Time (min.)
5x5	0.993 (±0.001)	0.975 (±0.009)	100%	0.435 (±0.1)	9.3
7x7	0.987 (±0.001)	0.984 (±0.002)	100%	0.208 (±0.043)	9.75
9x9	0.983 (±0.001)	0.981 (±0.001)	100%	0.15 (±0.029)	9.9
11x11	0.976 (±0.001)	0.972 (±0.001)	100%	0.122 (±0.018)	10.65

Table 4.7: Average Values (over 40 runs) for each distinct resolution for the Simple Tileset. (The numbers is brackets are the confidence intervals of the corresponding metrics).

Resolution	Final Best Fitness	Final Average Fitness	Success Rate (%)	KL Divergence	Time (min.)
5x5	0.999 (±0.001)	0.942 (±0.018)	100%	0.825 (±0.059)	11.55
7x7	0.993 (±0.002)	0.905 (±0.033)	90%	0.448 (±0.037)	11.7
9x9	0.987 (±0.002)	0.89 (±0.019)	55%	0.322 (±0.036)	12.45
11x11	0.973 (±0.002)	0.922 (±0.007)	7.5%	0.238 (±0.023)	14.1

Table 4.8: Average Values (over 40 runs) for each distinct resolution for the Medium Tileset. (The numbers is brackets are the confidence intervals of the corresponding metrics).

for a single image which in turn creates more complex images and thus leads to lower KL divergence. Lastly, the computational time needed for a single run to complete also increases the higher the resolution gets. That was to be expected too, since the image consists of more tiles that the algorithm has to go through every generation.

The increase in time needed combined with the decrease of the overall performance leads us to the conclusion that increasing the resolution of the input images above $5x5$ is not optimal. On top of that, using a higher resolution image as sample to generate a $20x20$ resolution image is not a very optimal approach since a situation where the input and the output image have similar resolutions does not make much practical sense. That is why in this approach the input images that will be used will have a resolution of 5 tiles by 5 tiles for the rest of the experiments.

4.6 Fitness Function Testing

On all the previous groups of experiments the focus was optimizing the generation of sample images that can be used as input for the WFC algorithm. The main goal so far was to create good quality images in terms of being flawless (no tiles being out of place) and complex in the sense that they provide some interesting features that would make a game more appealing to players. Apart from the constraint that was enforced on the fitness function - setting the

Resolution	Final Best Fitness	Final Average Fitness	Success Rate (%)	KL Divergence	Time (min.)
5x5	0.997 (±0.001)	0.714 (±0.031)	100%	1.685 (±0.075)	13.5
7x7	0.952 (±0.017)	0.522 (±0.038)	27.5%	1.13 (±0.052)	16.8
9x9	0.633 (±0.038)	0.247 (±0.025)	0%	0.738 (±0.043)	17.25
11x11	0.28 (±0.06)	0.058 (±0.042)	0%	0.721 (±0.064)	25.05

Table 4.9: Average Values (over 40 runs) for each distinct resolution for the Full Tileset. (The numbers is brackets are the confidence intervals of the corresponding metrics).

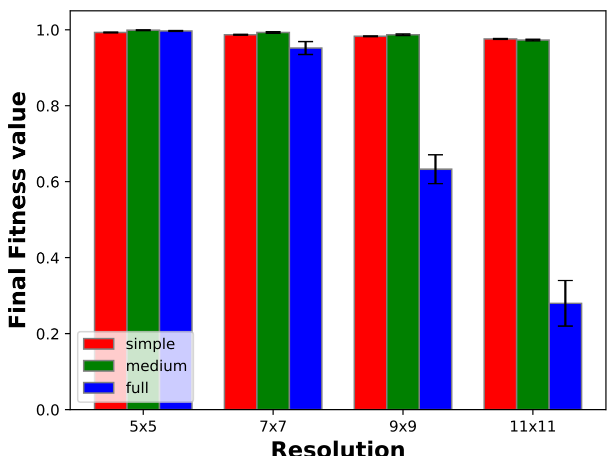


Figure 4.5: Final fitness average values (across 40 runs) for each resolution on all three tilesets (resolution numbers refer to tiles not pixels).

fitness function to zero if the WFC algorithm could not produce an output when using the phenotype image of the gene as input - our approach hasn't focused directly on optimizing the results generated through the WFC algorithm. That is why in this experiment we are going to modify the fitness function so that it takes into account the playability of the output images/maps.

The playability of a map can be calculated using different metrics, like traversability, balance or fairness [19, 13, 18], but in this experiment, we will only focus on the traversability part. Specifically, we will run a simple A* algorithm [3] on the map that is produced from the WFC algorithm to find the paths that connect the houses on the map with each other and calculate the percentage of paths that were calculated over the number of pairs of houses that exist on the map. This can be implemented because, as discussed on Section 3.1.1, each of the tiles holds information of whether it is walkable or not, which in turn allows A* to calculate the paths on the generated images. An algorithmic representation of this procedure can be seen on Algorithm 2, while on Figure 4.6 a visual representation of how the paths that are calculated look like is presented. This fitness is immediately set to zero if the map does not have two or more houses on it and has a maximum value of one if all the houses on the map are connected to each other.

Algorithm 2 Pixel Similarity Fitness

1: **for all** Houses **do**
2: Check if there is an A* path to all other houses on the image
3: **if** Path Exists **then**
4: $pathsFound = pathsFound + 1$
5: **end if**
6: $allHouseCombinations = (numberOfHouses * (numberOfHouses - 1))/2$
7: $fitness = pathsFound/allHouseCombinations$
8: **end for**

In this experiment the fitness of each gene derived from two separate functions, the pixel

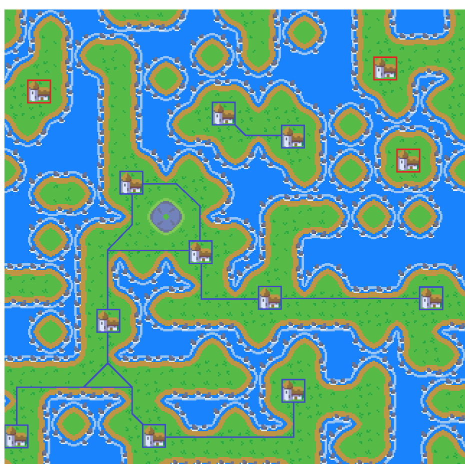


Figure 4.6: Visual representation of the paths that are calculated. The blue lines are the paths that have been found by the A* algorithm, while the houses that are highlighted in red are the ones that are not connected to any other house.

Fitness Function	Final Best Overall Fitness	Final Best Pixel Similarity Fitness	Final Best Traversability Fitness	Pixel Similarity Success Rate (%)	Travesability Success Rate (%)	KL divergence	Time (min.)
Medium Tileset							
Pixel Similarity	1.36 (±0.112)	0.999 (±0.001)	0.363 (±0.112)	100%	2.5%	0.83 (±0.06)	11.55
Pixel Sim. + Trav/lity	1.69 (±0.128)	0.998 (±0.001)	0.696 (±0.127)	100%	30%	0.7 (±0.05)	241.5
Full Tileset							
Pixel Similarity	1.21 (±0.1)	0.997 (±0.001)	0.215 (±0.099)	100%	2.5%	1.68 (±0.08)	13.5
Pixel Sim. + Trav/lity	1.28 (±0.129)	0.996 (±0.001)	0.281 (±0.128)	95%	7.5%	1.68 (±0.09)	162.6

Table 4.10: Average Values (over 40 runs) for each fitness function on the Medium and the Full Tilesets. (The numbers is brackets are the confidence intervals of the corresponding metrics).

similarity one that we have been using so far and the traversability one that was explained in this Section. However, the two fitness values are not independent to each other. We first make sure that the gene has a pixel similarity fitness value of one, before evaluating the output images that are generated by the WFC algorithm. This lowers the complexity of the fitness function and ensures that the output maps are both aesthetically "flawless" and playable, but changes the max fitness value that a gene can have to two, one for each of the two distinct fitness functions.

In particular, in this experiment the setup that was assumed to be the optimal one so far was used, which is: 50 generations, a population of 50 genes, a 5 tiles by 5 tiles phenotype image for each gene and the genetic operator 7 from Table 3.2 for the mutation of the genes. However, since the traversability fitness function would be utilized, the maps had to have houses on them, which are not present in the simple tileset. Therefore, for this experiment only the medium and the full tileset were used. In this part the fitness values of the final genes that we got from the previous experiments were recalculated, using the traversability fitness function, so that comparisons between the two methods could be drawn. For the recalculation we used the same random seeds that were used in the previous experiments and no new images were generated. This was done to ensure that the factor of randomness that derived from the seeds would not affect the results and lead us to false assumptions.

Apart from using a different fitness function, the random seeds are also getting evolved

this time. This means that for this experiment the genes that were selected and mutated did not get a new set of 10 random seeds that are used by the WFC algorithm, but instead every time a new gene is created it is assigned with random seeds that come from crossovering or mutating the random seeds of its parents. The probability for both the mutation and the crossover of the random seeds is 50% and the crossover method is a uniform crossover, while for the mutation method every random seed could be mutated with a 0.5% chance. When a random seed is mutated it is replaced with another randomly picked seed. Once the random seeds of both the parents are evolved, their genotype image is mutated too using the same procedure as before. In this experiment we also keep track of the traversability success rate, where a successful run is one where the map that was created is both flawless and fully traversable, meaning all the houses on it are connected to each other.

Table 4.10 features the results that we got for the medium and full tileset respectively. The first row of each subtable corresponds to the fitness recalculation of the previously generated images, while the second row represents the average values from running the evolutionary algorithm using the new fitness function. It is obvious that on the medium tileset our evolutionary algorithm with the new fitness is performing much better in terms of overall fitness. Specifically, the average pixel similarity is very close in both cases, but the average traversability fitness is significantly higher when the new fitness function is used. That was to be expected since the new fitness function directly aims on creating images that are both flawless and traversable when used as maps. In terms of KL divergence there is a small drop when using the traversability fitness function. The reason behind this could be the fact that our algorithm will naturally favor images that have house tiles on them and the medium tileset only has one house tile. This in turn can lead to the generation of images that have the same house tile multiple times, rather than several distinct tiles.

Unfortunately, the most significant difference between the two methods is that of the computational time to complete a single run. In particular, the algorithm had to run for 241, 5 minutes to complete a single run, which combined with its inability to create traversable maps of every run (30% success rate) makes the use of the traversability fitness function a non-optimal choice. The reason is that it is both unreliable in terms of providing useful results and the time that it takes to generate a single map is too long to be used for a design assisting tool. The same applies for the full tileset, where the results for both the fitness functions are really close to each other in terms of overall final fitness, final pixel similarity fitness, traversability fitness and KL divergence as shown on Table 4.10. The only significant difference again is the average time taken to complete a run which was 162, 6 minutes compared to the 13, 5 minutes a run using only the pixel similarity fitness takes. Combining it again with the very low success rate of 7, 5% on generating traversable maps and a 95% success rate on producing flawless images, leads to the assumption that using the traversability fitness function does not give the desirable results that were expected.

4.7 Expressivity Analysis

So far the experiments were focused on optimizing the generation of the sample input images that would be given as input to the WFC algorithm and the analysis was quantitative, aimed at maximizing the performance metrics that were established on Section 4.1. In this section, the expressive range of the algorithm will be explored and evaluated. To do this the images that were produced by the WFC algorithm when using the various sample images that our evolutionary algorithm generated as input will be used. This way, conclusions will be drawn

about the quality of the images that the WFC algorithm can produce using the setup that was established by the previous experiments.

The expressive range of a procedural generation algorithm as explained by Smith and Whitehead [37] is basically a classification of the style and variety of the content, in their case 2D platformer levels, that the generator can produce. In their paper, they discuss how such a quality can be measured and specifically for their project they introduce two metrics: linearity and leniency. Using these metrics they evaluate their generator and create heatmaps for the expressive range of their generator for each of the parameters that they tweak. However, these two metrics are related to the playability of platformer games' levels and are not useful in this approach, since the content that is created is images that should be useful as maps for top-down rpg-like games.

Some metrics that could be used are those introduced by Machado et al. [23] on their paper on how to computerize the measurement of visual complexity. In other words, in their work Machado et al. utilize theories that derive for psychology and Algorithmic Information Theory, to introduce three metrics that can be used to measure the visual complexity of an image. Specifically, these metrics are the edges of the image, which according to Palmer [28] are vital for human and thus computer vision, the compression size of the image, based on Salomon's theory [31] that simple images have redundant data and are easily compressed, and lastly the entropy estimations by Zipf's Law [44].

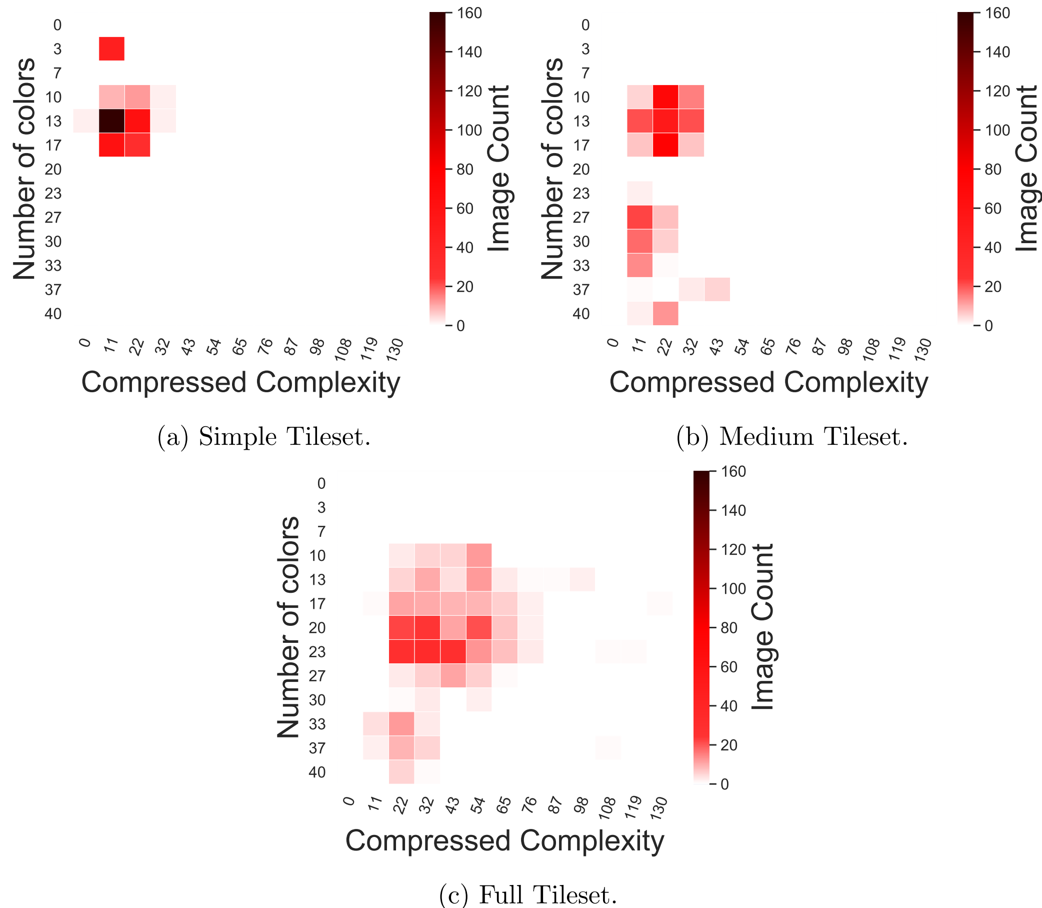
In this case the compression size and the number of distinct colors will be used as metrics for evaluating the complexity of the output images. Specifically, the images that are going to be evaluated are those produced by the WFC algorithm when the input images are those generated using the optimal parameters that were found through experimentation (50 generations, 50 genes, Genetic Operator 7, $5x5$ input images, pixel similarity fitness function). For each of the input images 10 output images of resolution 20 tiles by 20 tiles we generated, one for each of the random seeds of the gene. So at the end 400 images were evaluated (since each of the 40 runs produce one input and 10 output images) for each of the three tilesets.

The colors of each image were calculated as the distinct RGB colors of the image's pixels. In particular, we performed a simple merge of colors that are really close to each other on the RGB table with a tolerance of 10. That means that the colors whose R, G or B values had a difference lower than ten were considered to be the same color. The compression size of the images was calculated by using a simple JPEG compression algorithm. According to Machado et al. [23] the complexity estimate of an image, i, according to a lossy encoding scheme (in this case JPEG), f, is given by the following formula:

$$Complexity(i) = rmse(i, f(i)) * s(f(i))/s(i), , \tag{4.3}$$

where $rmse$ is the root mean square error between the original and the compressed image and s is a functions that calculates the file sized of an image. The details regarding the compression technique are not relevant for the scope of this thesis, so they will not be discussed further.

Figure 4.7 presents the results in terms of distinct colors versus the compressed complexity of the images in the form of heatmaps. For the simple tileset the images are highly concentrated on the top left corner of the heatmap. This means that the images produced using it are not very complex, since their compressed complexity values are relatively low and the colors that are featured in them are on the lower spectrum of the expressive range when it comes to color diversity. This can also be observed on Figure 4.8, where the image



(a) Simple Tileset.

(b) Medium Tileset.

(c) Full Tileset.

Figure 4.7: Our algorithm's expressive range in terms of number of colors and compressed complexity for each of the tilesets.

that corresponds to the lowest number of colors and compressed complexity is one made out of two tiles: a coast corner tile and a grass tile. Images like this one are frequently generated when using the simple tileset, which can lead to the assumption that this particular combination of tiles features a high pixel similarity. Unfortunately, such an image is not very interesting to be used as a map for a game and thus we favor images that feature a higher complexity. At the same time the images that correspond to the highest number of colors and the highest compressed complexity, which are featured on the same Figure 4.8, are relatively plain too, featuring a few more tiles than their lower counterparts and some more interesting patterns.

For the medium tileset the images are more spread out in terms of colors. As shown on Figure 4.7 the number of colors of the generated images are overall higher than the simple tileset, but their compressed complexity is not much higher. To be more precise, for both the tilesets the complexity of the images did not manage to get higher than 43, a value which is lower than the median value of the complexity spectrum. That was expected, if we think that the difference between the number of tiles from the simple to the medium tileset (6 tiles) is not very significant. This means that there weren't a lot of new colors added to the pallete of the algorithm and thus most of the images are still concentrated on the lower number of colors, while maintaining the simplicity that the images of the simple

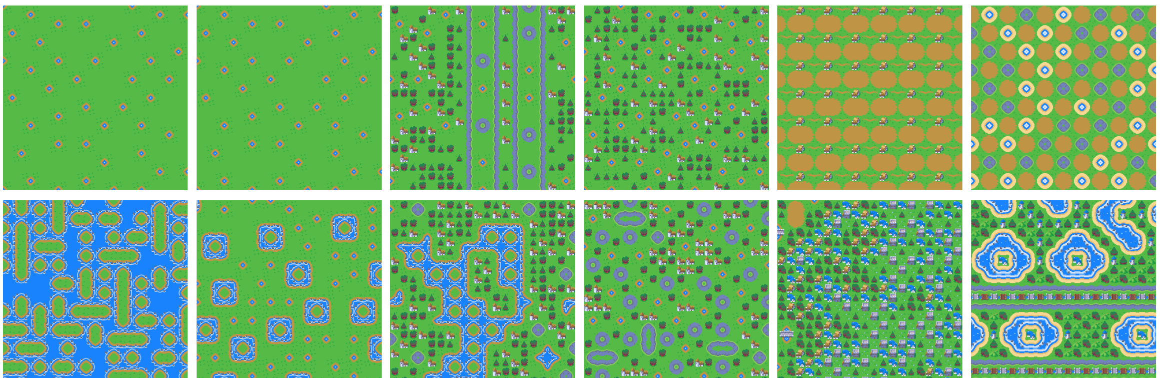


Figure 4.8: The generated images that feature the minimum (up) and maximum (down) values for the number of distinct colors and the compressed complexity for the simple (left four), medium (middle four) and full (right four) tilesets. The images on the left of each quartet correspond to the size, while the right correspond to the colors.

tileset featured. This is confirmed on Figure 4.8 where the images that correspond to the medium tileset, feature some new terrain types but they remain relatively simple, since there aren't many new interesting patterns on the images.

Finally, on the full tileset the images are much more spread out than the previous two tilesets. The generated images do not feature a larger amount of colors in general, which is expected considering that the tileset consists of many more tiles compared to the other two (65 new tiles compared to simple and 59 compared to medium). As a result the algorithm can now produce a much wider variety of results and use tiles that correspond to much more complex terrain types than before, generating images that feature much more interesting patterns and a higher diversity of terrain types. For example, on Figure 4.8 we can see that the images generated using the full tileset feature sandy beach tiles, bridges and castles which on the one hand are unique only to this tileset and on the other hand are much more complex tiles on their own compared to the grass tiles and the simple waterside tiles that the smaller tilesets have.

Apart from observing the difference between the complexity of the images that are generated using the three distinct tilesets, another very interesting thing to notice is that although most of the images are concentrated on specific points, the overall expressive range of the algorithm gets higher the larger the tileset gets. Specifically, for the full tileset, and perhaps the medium on, the images that are not part of the highest concentration points are spread around on various points of the expressive spectrum. This means that even when the implemented evolutionary algorithm is applied to the WFC algorithm the expressive capabilities of the algorithm are not reduced significantly. On the contrary, the algorithm can still produce a wide variety of images ranging from complex to simple and from colorful to less colorful, especially on larger tilesets.

4.8 Robustness

For all the experiments that were conducted so far in this section the same three tilesets that were discussed on Section 3.1.1 were used. The goal, however, is to implement a tool that designers could use universally, meaning that they shouldn't be bound to few specific tilesets. In this approach, the fitness function that was implemented was based on the

Tilesets	Game Genre	Number of Tiles	Resolution of Tiles
Tileset 1[1]	Platformer	55	16x16
Tileset 2[2]	Platformer	58	16x16
Tileset 3[3]	Overworld	198	16x16
Tileset 4[4]	Overworld	141	16x16
Tileset 5[5]	Overworld	1003	16x16

Table 4.11: Details about each of the tilesets used for the robustness testing.

pixel similarity of the adjacent tiles from which the image is created. This means that the evolutionary algorithm should not have a problem generating good results regardless of the tileset that is given to it, as long as the tiles can be compared to each other in a pixel by pixel manner. Furthermore, the WFC algorithm is not bound to a specific tileset either, but it is very dependent on the symmetry system that we discussed on Section 2.2.2.

4.8.1 Setup

In this group of experiments the robustness of the algorithm will be evaluated. Specifically, five new experiments will be conducted using a new tileset every time. Two of the tilesets that were picked contain tiles that are used to generate levels for 2D platformer games and the rest are overworld tiles that can be used to create levels or worlds for 2D role playing or adventure games. The experiments with these tilesets will test if the algorithm can produce good enough maps for 2D rpg games, which was the initial aim, using any overworld tileset and also if it can produce images that can be used as levels for other genres of games, like platformers. On Table 4.11 some details regarding each of the tilesets are provided.

Apart from establishing the tilesets that will be used, we also have to assign each of the individual tiles a symmetry type. As discussed at the end of Section 3.1.1 for the three tilesets that were used this procedure was done manually. Doing so again though would require an amount of time that was not available to us. The solution to this problem came from the fact that the symmetry types are only relevant for the WFC algorithm and do not interfere with the evolutionary process of our algorithm. This means that the robustness of the evolutionary algorithm can still be tested, without having to take into account the results produced from the WFC algorithm. That is why for each of the tiles on all of the tilesets used in this Section a symmetry type of “X” was assigned.

Other than that, the parameters of the evolutionary algorithm remained the same. The selection and the replacement methods were the ones discussed on Section 3.2, the algorithm ran for 50 generations and the population had 50 individuals at all times. The genetic operator was Operator 7 from Table 3.2 and the fitness function was the simple pixel similarity one.

[1] Source of the tileset: `https://rottingpixels.itch.io/platformer-dungeon-tileset`
[2] Source of the tileset: `https://grimfaith.itch.io/platformer-tileset`
[3] Source of the tileset: `https://szadiart.itch.io/craftland-demo`
[4] Source of the tileset: `https://beast-pixels.itch.io/overworld-tileset-grass-biome`
[5] Source of the tileset: `https://mattwalkden.itch.io/rpg-overworld-tileset`

Tilesets	Final Best Fitness	Final Average Fitness	Success Rate (%)	KL Divergence	Time (min.)
Tileset 1	0.999 (±0.001)	0.995 (±0.004)	100%	2.365 (±0.088)	2.4
Tileset 2	1 (±0.001)	0.976 (±0.006)	100%	1.427 (±0.05)	2.55
Tileset 3	0.98 (±0.005)	0.946 (±0.012)	100%	2.663 (±0.064)	4.95
Tileset 4	0.839 (±0.006)	0.8 (±0.01)	60%	2.515 (±0.096)	3.9
Tileset 5	0.993 (±0.002)	0.965 (±0.006)	100%	3.99 (±0.055)	6.6

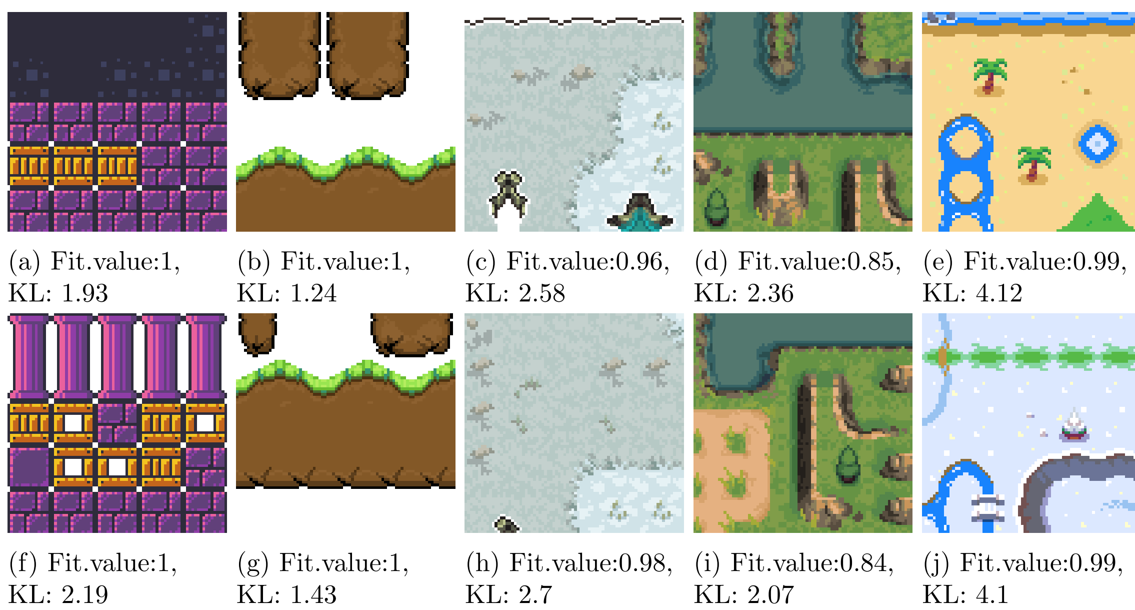
Table 4.12: Average Values (over 40 runs) for each of the tilesets that were used for the robustness testing.(The numbers is brackets are the confidence intervals of the corresponding metrics).

4.8.2 Results

For each of the five tilesets of Table 4.11 a full experiment of 40 runs was executed. Table 4.12 contains the results for each of them in terms of fitness, success rate and KL divergence. At this point, it is worth mentioning that the success rate in this case is measured based on the visual appearance of the image and not its playability or complexity. A successful run is one that generates an image that does not contain any tiles that are out of place in the sense that they are dissimilar to their neighboring tiles.

Based on the results produced, the algorithm was relatively successful, managing to have a 100% success rate on most of the tilesets apart from tileset 4, where it reached 60%. The struggle of the algorithm to produce good results when using tileset 4 can also be seen on the final fitness values for the same tileset. Specifically, the final best and average values for tileset 4 are much lower than those of the rest of the tilesets. However, an interesting observation on this one is that the final fitness values were relatively low regardless if the generated image was flawless or not. That could mean that the tiles of this tileset might seem to fit together on the naked eye, but a pixel by pixel comparison shows otherwise. As a matter of fact, if we examine the two images that we present on Figure 4.9 we can see that the images seem flawless, having no tile that does not fit with its neighbors, but their fitness values are relatively low. This could be caused by minor pixel dissimilarities between tiles that should theoretically fit with each other.

Another interesting comparison in this group of experiments is that of tileset 1 against tileset 2. Both these tilesets can be used for platformer games and our algorithm managed to have a very good performance on both of them. However, there were some differences between the images that were created for each of them. In particular, for tileset 1 as shown on Figure 4.9 there are tiles on the images that do not look like they match their adjacent tiles in terms of pixel color similarity. For example, there are some yellow tiles next to some purple ones. However, the fitness of both these images is 1 meaning that all the tiles on the image are a perfect fit with its other. That could be caused by the fact that some of the tiles could be framed in a transparent or a black box, making it so that our algorithm detects no difference on the pixels at the edges of these tiles. This in turn, leads to the algorithm creating more wall-like images like the bottom one featured on Figure 4.9 for tileset 1, rather than ramp patterns on which the player can move, like the top one on the same Figure. On the other hand, on tileset 2 most of the generated images were platforms. In this case, most of the tiles on the tileset were transparent for most of their parts. This means that there was a lot of negative space which the algorithm could easily match with other transparent tiles. If we combine this with the fact that most of the ramp tiles are



(a) Fit.value:1, KL: 1.93 (b) Fit.value:1, KL: 1.24 (c) Fit.value:0.96, KL: 2.58 (d) Fit.value:0.85, KL: 2.36 (e) Fit.value:0.99, KL: 4.12

(f) Fit.value:1, KL: 2.19 (g) Fit.value:1, KL: 1.43 (h) Fit.value:0.98, KL: 2.7 (i) Fit.value:0.84, KL: 2.07 (j) Fit.value:0.99, KL: 4.1

Figure 4.9: Two examples of generated images for each of the five tested tilesets, starting from tileset 1 on the left and ending at tileset 5 on the right.

transparent for the majority of their image, the algorithm can create platforms easier than plain walls.

As for the rest of the tilesets, some of the generated images on Figure 4.9 and the overall results did not feature any specific issue or any interesting observations. Although the input images that the evolutionary algorithm produced for each of the tilesets were mostly flawless, the same can't be said for the images that were produced by the WFC algorithm. That was to be expected though, since all tiles were assigned a symmetry type of "X". This in turn means that the WFC algorithm did not have any information about how these tiles should be rotated and thus the images that it generated look like randomly placed tiles in all slots.

Chapter 5

Conclusion

This Chapter provides a summary of the results that were produced from the experimentation phase of this implementation. Following that the answers for each part of the research question are given and then some of the limitations that we came across along with ways to resolve them are discussed. Finally, some ideas that could potentially expand on this work and improve on it are suggested.

5.1 Results summary

Based on the results gathered from the experiments this approach was overall successful. We managed to combine an evolutionary algorithm and the Wave Function Collapse algorithm to a considerable degree of success, without running into a lot of complications with the core functionality of the WFC algorithm. This implementation was able to evolve the patterns that can be used as input for the Wave Function Collapse, producing very promising results in terms of performance, robustness and efficiency. Furthermore, through the experimentation and the fine-tuning of this algorithm we got a very good understanding of the complexity of the WFC algorithm and the parameters that can have the highest impact of the produced results, as well as the evolutionary parameters that are the most important when it comes to manipulating it.

To start with, from a perspective of performance the evolutionary algorithm managed to have a very consistent performance throughout all the experiments, performing very well in terms of fitness when the aim was to create images that featured high pixel similarity between their tiles. Specifically, we can assume that the algorithm works really well on small tilesets, since for the simple tileset (5 unique tiles) it consistently averaged fitness values of $0,9$ or higher. The case was similar for the medium tileset (11 unique tiles), although for this tileset the algorithm was struggling to perform well when the overall stochasticity was really high. Nonetheless, it still managed to average high fitness values throughout the experimentation phase when the right parameters were applied. This in turn, means that applying methods that are stochastic in their most parts to generate results is not the optimal approach when evolving images. On the contrary, methods that featured controlled stochasticity accompanied by some deterministic elements performed the best and that was the reason why they were favored in this approach.

The importance of the genetic operator used can be clearly seen on the results for the full tileset (70 unique tiles), which is an example of a larger scale tileset. In particular, when testing the genetic operators, deterministic repair methods were applied, combined

with more sophisticated selection and removal methods in order for the algorithm to be able to perform at a decent level and be able to produce some good results both in terms of fitness and in terms of visual flawlessness. Before that the algorithm was averaging fitness values below $0,8$ with a massive increase on the last two genetic operators that were used for an average of $0,95$ final fitness and a success rate of 75% and 90% respectively. Speaking of success rates, the results produced on that front were similar for the rest of the tilesets too, where the operators that were using stochastic methods failed to create flawless results completely, even though the averaged high fitness values, and the ones that were more deterministic were able to have a perfect success rate in most of their runs. However, the factor that led to the final decision on the genetic operator was that of the KL divergence and more specifically the high KL divergence of operator 8, which had the best overall performance. The high KL divergence meant that although the images were visually flawless, they were generated using only a very small percentage of the overall tileset, which in turn meant that those images were lacking in complexity or interesting patterns. That is what led to the decision to proceed with the next best option which was operator 7 (see Table 3.2 for references).

Although the genetic operator used to evolve the images was the most important parameter in all of the experimentation phase, the size of the population was also a very important factor, especially when it comes to larger tilesets. Specifically, as shown on Table 4.6 for the full tileset the algorithm was generating perfect results consistently for population sizes of 50 or more. The same can't be said though for the size of the images. Specifically, our implementation is having a difficult time performing on a good level the higher the resolution of the sample images that are getting evolved gets. This can either mean that the algorithm needs more computational time to generate a good result of higher resolution or that a more sophisticated genetic operator needs to be applied to improve its performance. However, in the context of this thesis there was no need for a higher resolution sample image, since the output image resolution stays the same, at $20x20$ for the whole duration of our experimentation phase. Higher resolution sample images would be useful only if the resolution of the output images was higher too, since in that case there would be a need for more patterns in order to create interesting images.

The algorithm proved to be very efficient in creating input images that consisted of tiles that fitted with each other in terms of color. Although this ensures that the images generated by the WFC algorithm will be flawless too, it does not guarantee that they will be playable if they are used as maps. This is what is evaluated of Section 4.6 by utilizing a fitness function that assigns a fitness value to every gene based on the traversability of the output maps generated by the WFC algorithm with that specific gene as input. As we can see on the results on Table 4.10 the performance of the evolutionary algorithm in terms of traversability were not very promising. Specifically, the algorithm managed to maintain its good performance when it comes to producing flawless images, but the maps generated by the WFC algorithm were not fully traversable most of the time. That combined with a massive increase in computational time needed to complete a full evolutionary loop suggests that the approach on the playability of the maps was nowhere near the optimal one.

Even though the algorithm was not able to create fully traversable maps, it was still able to create a variety of images as shown on Figure 4.7. The expressivity range of the algorithm, meaning its ability to create a wide range of diverse images, is quite spread across the spectrum of possible results. There are of course some images that are concentrated in specific points of the expressive spectrum and the overall complexity of the images increases the larger the tileset gets. However, each of the tilesets also managed to create diverse

results, whether they were complex images when using a small number of tiles or simple images when using a larger number of tiles.

Lastly, the algorithm proved to also be relatively robust when tested on five randomly selected tilesets. Specifically, its performance in terms of fitness was very good for almost all of the tilesets that we used, apart from one. On this one, the algorithm seemed to struggle a bit both to reach high fitness values and generate flawless results. The reason behind this might have been either our algorithm's inability to perform well when using this tileset, or perhaps the tiles of this tileset were not very similar to each other on a pixel by pixel comparison. However, the promising results we got on the rest of the tilesets leads us to believe that our implementation is robust enough to be a helpful tool for game designers. The only problem in this case is the WFC algorithm itself and more precisely the symmetry system that comes with it. The reason being the fact that all the tiles have to be assigned the correct symmetry type in order for the WFC algorithm to work properly, which if the tileset is large can be a very time consuming task.

5.2 Addressing the research question

Regardless of the good performance of the implementation, we need to ensure that the algorithm managed to answer all four parts of the research question. In this Section the answers are discussed one by one:

- **Can evolutionary algorithms be applied to the WFC algorithm in order to control the stochasticity of the outputs produced by it, without requiring many computational resources?** The answer for this question can be separated into two parts. For the first part, an evolutionary algorithm can be combined with the WFC algorithm and does not have a negative impact on the results that are being generated as long as the right parameters are set. The implemented algorithm in this thesis was really successful in generating good sample images that were visually flawless and thus the WFC algorithm was able to generate images that feature the same patterns as the input images without any further complication. When it comes to controlling the stochasticity of the outputs of the WFC algorithm, the algorithm unfortunately did not manage to be successful. Specifically, our approach on evaluating the playability of the output images through the traversability of the map did not produce the desired results. That being said, there was definitely an improvement when we altered the fitness function so that it favors images that lead in the generation of more traversable images, which leads to the assumption that a more sophisticated fitness function will be able to ensure the playability of the generated images.

- **Which parameters are the most important in determining an optimal setup for the evolutionary algorithm?** Taking into consideration the results gathered at the end of the experiments we could assume that although the setup that was used might not have been the most optimal one, it was definitely one that could produce good results. In Chapter 3, we went through some tools that were either used unchanged or that were implemented specifically for this approach. The tools that were not implemented from scratch were the core functionality of the WFC algorithm and Parker's implementation [29] on generating rulesets based on sample images. Then, some parts of the evolutionary algorithm that were assumed would work well on this approach were established, like the selection and the replacement method for

which we used a roulette selection technique and an elitism method respectively. The fitness function was set as a pixel similarity fitness function. The idea behind this is that images that are flawless in the sense that they are made of tiles that fit with each other visually and are not dissimilar from their adjacent tiles, will lead to the generation of similarly flawless images when used as input for the WFC algorithm. That was indeed the case as images that contained no “out of place” tiles, produced images that also featured this trait. On the fourth group of experiments an attempt to use a different fitness function was made, but its poor performance locked in the pixel similarity as the fitness function of choice. The genetic operator and the size of the population were established through the first two groups of the experiments. The final genetic operator that was chosen was one that would select the tile to change on the image via a roulette selection technique, remove it along with its Von Neumann neighboring tiles and then repair the empty slots deterministically, meaning that it filled each slot with the tile that would result in the best fitness value. The size of the population was determined to be 50 as it was the one that offered the best trade-off between performance and computational time. The number of generations was set initially on 50 and remained like this as the results were already very good. Finally, the resolution of the input images that are evolved by the algorithm was set to $5x5$ (5 tiles by 5 tiles), since based on the results gathered from the third group of experiments, the algorithm was struggling to generate flawless images for the full tileset the higher the resolution got. All the parameters that were presented above are basically the ones that were established to be the optimal in this case and are the answer to this specific question.

- **Are the results provided using this method diverse?** To evaluate this we used the KL divergence and the expressivity analysis that was conducted at the end of the experimentation. Although the KL divergence is a very good tool to give an idea of the complexity and diversity of the sample images that the algorithm was creating, it only proved useful when used to compare images of the same tileset. The reason behind this is that the combination of using larger tilesets and keeping the same resolution for all the tests was naturally producing higher KL divergence values the larger the tilesets were getting. It did, though, give us a good perspective of the complexity of the results produced on the first and second group of experiments, where images that were produced from the same tileset had to be compared. The KL divergence however was used only to evaluate the sample input images generated by the evolutionary algorithm, but not the images produced by the WFC algorithm. For that we did an expressivity analysis on all the images that were produced for each of the tilesets through the WFC algorithm and based on the results that are showcased on Figure 4.7 our approach managed to produce a wide range of images both in terms of complexity and in terms of diversity.

- **Is the evolutionary algorithm robust enough to work on any given tileset?** The short answer here would be yes. However, the robustness of the algorithm seems to be bound to the nature of the tileset that is being used. This means that if the tileset that is used, doesn't feature color similarities between the tiles that are supposed to be placed next to each other, it might have difficulties generating visually flawless images. Other than that the implemented algorithm performed really well when random tilesets where used regardless the size of the tileset or the resolution of

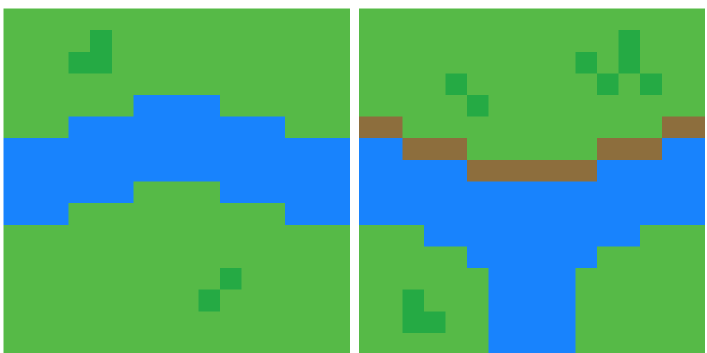


Figure 5.1: Two tiles that have slight dissimilarities on their edges.

the tiles or even the genre of the game that the tileset was meant for.

5.3 Limitations

This Section presents some known limitations of the methodology along with some potential solutions that future attempts could use to address them.

5.3.1 Tileset

The first and most important of the limitations had to do with the tileset used in this thesis. Specifically, the tileset itself was causing some issues that had to be resolved in order for the algorithm to work optimally. One of those issues was the fact that some of the tilesets that were supposed to be placed next to each other did not feature the appropriate pixel similarity. This was caused because parts of the edges of the two adjacent tiles were dissimilar, resulting in lower fitness values than intended. An example of such a case is shown on Figure 5.1. The solution that was used to solve this issue was to "correct" the faulty parts of the images. By that we mean that we had to go through every tile that we used on this tileset and change the colors of their edges so that they would be a perfect match with the tiles that they were supposed to be perfect matches with. This is obviously not an optimal way to address this problem since the task of correcting all the tiles is very time consuming. The best solution here would be to implement an algorithm that could work around this problem, by doing either a more generic comparison rather than direct pixel by pixel one or checking multiple pixels at the same time rather than one at a time.

Another issue that was caused by the nature of the tileset is when two tiles were not meant to be adjacent but were still placed next to each other by the algorithm because they had high pixel similarity between them. Such an example is shown on Figure 5.2. In cases like these there was a slim chance that the algorithm would get confused and place those two tiles next to each other, assuming that they are the best match. Although, most of the time this problem was getting resolved by the algorithm itself on the next generation, sometimes the algorithm would be stuck in a situation where the tile that needed to be placed in that position would not fit the rest of the image and thus was not selected as a replacement. In this thesis, this issue was not addressed since its rarity did not impact the results to a great extent, but a good solution would be to use an even more sophisticated selection and removal procedure for the mutation of the images than the ones used here.

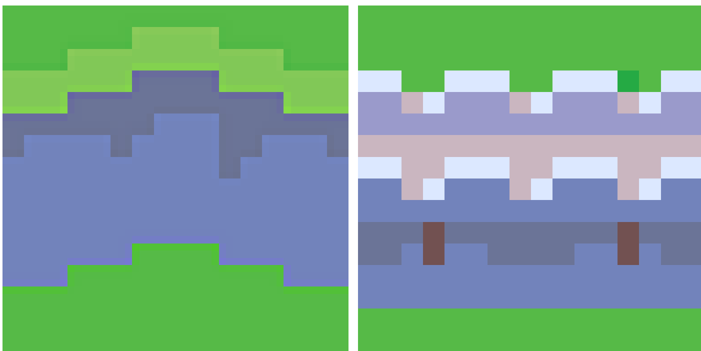


Figure 5.2: Examples of tiles that have some similar pixels but do not fit each other.

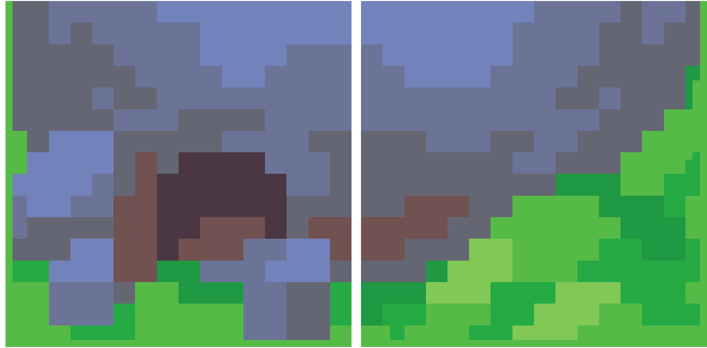


Figure 5.3: Two examples of tiles that can't be assigned a symmetry type.

Lastly, in the tileset there were several tiles that would not match any of the symmetry types presented by Gumin and assigning any of the types to them would have a negative effect on the images that were generated by the WFC algorithm. In Figure 5.3 some examples of such tiles are presented. To solve this issue these problematic tiles were removed from the tileset. A more appropriate solution would be to pre-propagate the output image when one of these tiles appeared and assign the appropriate tiles for their adjacent tiles so that the final image will not have any wrong tiles in the wrong place.

5.3.2 Ruleset generation

Another limitation was that of the generation of the rulesets. Although Parker's [29] implementation did not cause any direct issues, we believe that there is a more optimal way to produce a ruleset based on a sample image. The way that Parker implemented it was to create an adjacency rule for every pair of tiles that existed in the image and then create more rules based on the rotated version of the same tiles. This approach might have led to an excess amount of rules that might create contradictions when applied to the WFC algorithm. A more optimal approach to generating the ruleset would be to treat the image created by the evolutionary algorithm as the sample image of the Overlapping WFC model rather than the Simple Tiled model and avoid the complications that might occur when using a ruleset to establish the adjacency rules.

5.3.3 Simple Tiled model

Last but not least, another limitation was that only the SimpleTiled model of the WFC algorithm was utilized. Since we had to work with tiles the first option that came to mind was using the Simple Tiled model. However, our approach was to create sample images that would then be given to the WFC algorithm as input, so in theory the Overlapping model could be used to do that, since the whole procedure was more appropriate for it. Unfortunately though, the Overlapping model requires images of smaller size than the ones created using our tilesets, so swapping to it would require image conversions that could potentially reduce the quality of the input images to such an extent that the patterns that we wanted to have on them would be lost.

5.4 Future Work

In this section methodologies that could be applied to the same problem and could potentially perform similarly or even better than this approach will be proposed, as well as some changes that could perhaps enhance the performance of the algorithm even further.

5.4.1 Automated symmetry system

One of the most important issues in this thesis was the symmetry system that Gumin established for the tiles of the Simple Tiled WFC model. This symmetry system was implemented by Gumin to prevent the rulesets from being extensively lost, but it is a problem when it comes to large tilesets. Specifically, the fact that the user has to go through every single unique tile and assign a symmetry type to it can be a very time consuming task. However, this task can not be skipped since the WFC algorithm will not generate the correct results if the symmetry types of the tiles are not correct. Thus an algorithm that would automatically go through each of the tiles of the tileset and recognize which of the available symmetry types is the best for them would save the users some valuable time and complete the initial goal of this thesis on implementing a universal tool that game designers can immediately use to generate images with them having to barely get involved in the process. Such an algorithm could utilize image processing techniques to search and recognize patterns on the image of the tile that correspond to patterns that usually refer to one of the symmetry types and assign that tile to that type.

5.4.2 Alternative applications of evolution on Wave Function Collapse

This approach is focused on evolving the input images that we would use as input patterns for the WFC algorithm in order to generate images that could be used as maps for 2D role playing or adventure games. However, the WFC algorithm features a plethora of elements that could potentially get evolved instead of the input patterns. Some interesting approaches could be evolving the ruleset directly rather than evolving the sample image and then generating the ruleset based on that image or evolving the weights of the tiles. The weight of the tile refers to its probability to get picked as a tile to fill in an empty spot of the output image. Evolving the weights of the tiles might be a very good option when trying to control the outputs of the WFC algorithm, since manipulating them can have a direct impact on the way WFC works to produce the images.

Apart from evolving different elements of the WFC algorithm, another approach would be to implement a different fitness function. Specifically, in this approach the fitness function

was mostly aimed on creating good input images with little focus on the images that the WFC algorithm would generate. An interesting alternative here would be a fitness function that instead of evaluating the genes based on input images, it would apply the Map Sketched approach of Liapis et al. [19] to evaluate the playability of the output maps and assign the fitness value accordingly. Furthermore, a fitness function focused on creating input images that are playable themselves rather than visually flawless, could have greater control over the playability of the WFC algorithm's generated images.

Finally, although in the experiments a decent amount of genetic operators that can be used to evolve the input images were tested, there are still plenty of combinations and methods that could produce the same or even better results and perhaps do it faster as well. Some interesting options here would be to use a removal technique that removes the chosen tile and its Von Neumann neighbors of range 2 or to use a selection method that would select tiles from the image based on both their pixel similarity and frequency, so the tiles that appear more often would be favored. The use of such a genetic operator could overcome some of the issues that were discussed on the previous Section regarding the tileset or perhaps an operator that mutates a larger amount of tiles at the same time would be able to generate input images of higher resolution with more success than the implemented algorithm.

5.4.3 Mixed Initiative Approaches

This thesis' goal was to implement an application that would make level designer's life easier, by removing the need of generating the levels of a game by hand. Although this indeed could save up a lot of time during the development of a game, sometimes the algorithm does not manage to meet the designer's specific expectations. This is where mixed-initiative approaches come into play. An evolutionary algorithm that would evolve input image patterns guided by the designer's input could massively increase its performance. Specifically, the algorithm could start by evolving a specific set of patterns and at the end of each generation, the images that were generated so far could be presented to the designer. Then, he/she could pick which of these images meet some of the criteria that he/she established and the algorithm will base the fitness of the images on the selections of the designer. This way, the designer can have some control over the whole procedure and favor the images that would result in the generation of images that feature specific visual patterns or specific playability traits.

5.4.4 Going 3D

Lastly, the implemented algorithm evolves 2D images that are basically tilemaps. A very interesting expansion on this work would be to use it as a starting point to create an application that would have the same functionality but in 3D or to just apply it on games. Specifically, the algorithm is robust enough to have a good performance on any given input when it comes to 2D. With a few modifications the same algorithm could potentially be used to generate 3D content. However, the main focus of such an approach would be to come up with a way to determine the playability, since 3D maps can be much more complex than their 2D equivalents.

In the same mindset, this approach could be applied to some games that are already known to be utilizing the WFC algorithm, like for example "Bad North" (Plausible Concept, 2018) or "Caves of Qud" (Freehold Games, 2015). Both these games use specific design

constraints to ensure that the content generated by WFC is playable or at least usable in their respective cases. This thesis' implementation could be used to replace or enhance these constraints and evolve the content that will be used as input for WFC, in order to maximize their usefulness for the game. This way, the designers would only have to worry about creating the parts, whether they are tiles or blocks, that will create the initial content that will be then evolved by the evolutionary algorithm, in order to be then used as input for the WFC algorithm.

Bibliography

[1] A. Bassi, K. Lochan, S. Satin, T. P. Singh, and H. Ulbricht. Models of wave-function collapse, underlying theories, and experimental tests. *Reviews of Modern Physics*, 85(2):471, 2013. — Cited on page 11.

[2] P. Bromiley, N. Thacker, and E. Bouhova-Thacker. Shannon entropy, renyi entropy, and information. *Statistics and Inf. Series (2004-004)*, 2004. — Cited on page 11.

[3] X. Cui and H. Shi. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130, 2011. — Cited on page 39.

[4] I. M. Dart, G. De Rossi, and J. Togelius. Speedrock: procedural rocks through grammars and evolution. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, pages 1–4, 2011. — Cited on page 10.

[5] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial networks. *arXiv preprint arXiv:1406.2661*, 2014. — Cited on page 11.

[6] J. Grasso. Playing with sound: A theory of interacting with sound and music in video games. *The Bulletin of the Society for American Music*, 43(1):11, 2017. — Cited on page 8.

[7] D. Gravina and D. Loiacono. Procedural weapons generation for unreal tournament iii. In *2015 IEEE Games entertainment media conference (GEM)*, pages 1–8. IEEE, 2015. — Cited on page 8.

[8] M. Gumin. Wavefunctioncollapse. `https://github.com/mxgmn/WaveFunctionCollapse`, 2016. — Cited on pages iii, 1, 11, 12, and 13.

[9] P. J. Hancock. Selection methods for evolutionary algorithms. In *Practical Handbook of Genetic Algorithms*, pages 67–92. CRC Press, 2019. — Cited on page 17.

[10] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 9(1):1–22, 2013. — Cited on page 8.

[11] S. Hesterman and G. McAuliffe. Introducing zentangle in the early years. *Curriculum and Teaching*, 32(2):61–88, 2017. — Cited on page 16.

[12] L. Johnson, G. N. Yannakakis, and J. Togelius. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pages 1–4, 2010. — Cited on page 10.

[13] A. Khalifa, D. Perez-Liebana, S. M. Lucas, and J. Togelius. General video game level generation. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 253–259, 2016. — Cited on page 39.

[14] S. O. Kimbrough, G. J. Koehler, M. Lu, and D. H. Wood. On a feasible–infeasible two-population (fi-2pop) genetic algorithm for constrained optimization: Distance tracing and no free lunch. *European Journal of Operational Research*, 190(2):310–327, 2008. — Cited on page 19.

[15] D. P. Kingma and M. Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013. — Cited on page 11.

[16] M. Kleineberg. Wave function collapse city. `https://marian42.itch.io/wfc`, 2019. — Cited on page 15.

[17] A. Krolikowski, S. Friday, A. Quintanilla, and J. Schrum. Quantum zentanglement: Combining picbreeder and wave function collapse to create zentangles®. In *International Conference on Computational Intelligence in Music, Sound, Art and Design (Part of EvoStar)*, pages 49–65. Springer, 2020. — Cited on pages 4 and 16.

[18] A. Liapis, H. P. Martinez, J. Togelius, and G. N. Yannakakis. Adaptive game level creation through rank-based interactive evolution. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8. IEEE, 2013. — Cited on page 39.

[19] A. Liapis, G. Yannakakis, and J. Togelius. Towards a generic method of evaluating game levels. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 9, 2013. — Cited on pages 39 and 56.

[20] A. Liapis, G. N. Yannakakis, M. J. Nelson, M. Preuss, and R. Bidarra. Orchestrating game generation. *IEEE Transactions on Games*, 11(1):48–68, 2018. — Cited on page 7.

[21] A. Liapis, G. N. Yannakakis, and J. Togelius. Computational game creativity. ICCC, 2014. — Cited on pages 1 and 7.

[22] S. M. Lucas and V. Volz. Tile pattern kl-divergence for analysing and evolving game levels. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 170–178, 2019. — Cited on page 30.

[23] P. Machado, J. Romero, M. Nadal, A. Santos, J. Correia, and A. Carballal. Computerized measures of visual complexity. *Acta psychologica*, 160:43–57, 2015. — Cited on page 42.

[24] M. Mateas and A. Stern. Procedural authorship: A case-study of the interactive drama façade. *Digital Arts and Culture (DAC)*, 61, 2005. — Cited on page 8.

[25] J. Neumann, A. W. Burks, et al. *Theory of self-reproducing automata*, volume 1102024. University of Illinois press Urbana, 1966. — Cited on page 10.

[26] T. Nordvig Møller, J. Billeskov, and G. Palamas. Expanding wave function collapse with growing grids for procedural map generation. In *International Conference on the Foundations of Digital Games*, pages 1–4, 2020. — Cited on pages 4 and 16.

[27] J. Olsen. Realtime procedural terrain generation. 2004. — Cited on page 10.

[28] S. E. Palmer, K. B. Schloss, and J. Sammartino. Visual aesthetics and human preference. *Annual review of psychology*, 64:77–107, 2013. — Cited on page 42.

[29] J. Parker. Unity wave function collapse. `https://selfsame.itch.io/unitywfc`, 2016. — Cited on pages 4, 16, 22, 23, 51, and 54.

[30] M. Ranjitha, K. Nathan, and L. Joseph. Artificial intelligence algorithms and techniques in the computation of player-adaptive games. In *Journal of Physics: Conference Series*, volume 1427, page 012006. IOP Publishing, 2020. — Cited on page 9.

[31] D. Salomon. *Data compression: the complete reference*. Springer Science & Business Media, 2004. — Cited on page 42.

[32] A. Sandhu, Z. Chen, and J. McCoy. Enhancing wave function collapse with design-level constraints. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, pages 1–9, 2019. — Cited on pages 4, 13, and 15.

[33] J. Secretan, N. Beato, D. B. D Ambrosio, A. Rodriguez, A. Campbell, and K. O. Stanley. Picbreeder: evolving pictures collaboratively online. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1759–1768, 2008. — Cited on page 16.

[34] N. Shaker, A. Liapis, J. Togelius, R. Lopes, and R. Bidarra. Constructive generation methods for dungeons and levels. In *Procedural Content Generation in Games*, pages 31–55. Springer, 2016. — Cited on page 8.

[35] N. Shaker, J. Togelius, and M. J. Nelson. *Procedural content generation in games*. Springer, 2016. — Cited on pages 7, 8, and 9.

[36] M. Sicart. Defining game mechanics. *Game Studies*, 8(2), 2008. — Cited on page 8.

[37] G. Smith and J. Whitehead. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pages 1–7, 2010. — Cited on page 42.

[38] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius. Procedural content generation via machine learning (pcgml). *IEEE Transactions on Games*, 10(3):257–270, 2018. — Cited on page 10.

[39] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelbäck, and G. N. Yannakakis. Multiobjective exploration of the starcraft map space. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 265–272. IEEE, 2010. — Cited on page 19.

[40] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011. — Cited on page 8.

[41] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011. — Cited on page 9.

[42] G. N. Yannakakis and J. Togelius. *Artificial intelligence and games*, volume 2. Springer, 2018. — Cited on pages iii, 7, 9, 17, and 18.

[43] X. Yao and P. Darwen. Genetic algorithms and evolutionary games. *Commerce, Complexity and Evolution*, 313:333, 2000. — Cited on page 19.

[44] G. K. Zipf. *Human behavior and the principle of least effort: An introduction to human ecology*. Ravenio Books, 2016. — Cited on page 42.