Christian Wustrau

Search-based Procedural Content Generation with Rolling Horizon Evolutionary Algorithm for Tile-based Map Generation

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

INF FAKULTÄT FÜR INFORMATIK

Intelligent Cooperative Systems
Computational Intelligence

Search-based Procedural Content Generation with Rolling Horizon Evolutionary Algorithm for Tile-based Map Generation

Master Thesis

Christian Wustrau

August 23, 2022

Supervisor: Prof. Dr. Sanaz Mostaghim

Advisor: Dr. Christoph Steup

Christian Wustrau: *Search-based Procedural Content Generation with Rolling Horizon Evolutionary Algorithm for Tile-based Map Generation*
Otto-von-Guericke Universität
Intelligent Cooperative Systems
Computational Intelligence
Magdeburg, 2022.

Abstract

The generation of content for video games falls within the realm of human creativity and therefore poses an interesting challenge for computer methods to emulate this process. Procedural Content Generation in Games is a popular and well-studied general approach to this mentioned challenge and has originated many different forms to address this, most notably Search-based Procedural Content Generation (SbPCG). Rolling Horizon Evolutionary Algorithm (RHEA) are a subclass of Evolutionary Algorithms (EAs) capable of online decision making, which means they search for the best action to take during the game in a limited short amount of time. By combining Search-based Procedural Content Generation and fundamental ideas from Rolling Horizon Evolutionary Algorithms to enable online content generation with fairness, this work presents a novel approach to map generation that is applied to a tile-based open-source game as a proof of concept.

Contents

List of Figures V

List of Tables VII

List of Acronyms IX

1 Introduction 1

1.1 Motivation . . . 1
1.2 Research Questions . . . 3
1.3 Thesis Structure . . . 3

2 Background and State of the Art 5

2.1 Procedural Content Generation . . . 5
2.1.1 Categories . . . 8
2.2 Evolutionary Algorithms . . . 11
2.2.1 Building Blocks of Evolutionary Algorithms . . . 14
2.2.2 Rolling Horizon Evolutionary Algorithms . . . 19
2.2.3 Search-based Procedural Content Generation . . . 20
2.3 Tile-based Games . . . 23
2.3.1 Hexagonal Grids . . . 24
2.3.2 The Battle for Wesnoth . . . 28

3 Content Generation Rolling Horizon Evolutionary Algorithm 33

3.1 Concept and Overview . . . 33
3.1.1 Concept . . . 33
3.1.2 Desirable properties . . . 34
3.1.3 Overview . . . 36
3.2 Content Representation . . . 38

3.3 Fitness Functions . . . 39
3.3.1 Differential Tile Fitness . . . 40
3.3.2 Layout Entropy Fitness . . . 44
3.3.3 Village Fitness . . . 45
3.3.4 Weighted Sum . . . 48
3.4 Rolling Horizon Evolutionary Algorithm . . . 48
3.4.1 Evolutionary Algorithm . . . 48
3.4.2 Rolling Horizon . . . 50

4 Evaluation 53
4.1 Hypotheses . . . 53
4.2 Experimental Setup . . . 53
4.2.1 Data Acquisition . . . 53
4.2.2 Game Setup . . . 54
4.2.3 Metrics and Hyperparameters . . . 54
4.3 Experiments . . . 56
4.4 Results . . . 58
4.4.1 Hyperparameter Sets . . . 58
4.4.2 Hyperparameter Performance . . . 63
4.4.3 Turn-based Performance . . . 65
4.4.4 Generation-based Performance . . . 67
4.4.5 Speed Analysis . . . 69
4.4.6 Fairness . . . 71
4.5 Discussion . . . 73

5 Conclusion 75

Bibliography 77

List of Figures

2.1 Visualization of the basic process of Evolutionary Algorithms. . 13
2.2 Comparison of hexagonal alignment of vertical columns and horizontal rows. 24
2.3 Overview of hexagonal coordinate system layouts. 25
2.4 Cube and axial coordinate system. 26
2.5 First and second level neighborhood with $N_1 = \{1,2,3,4,5,6\}$ and $N_2 = \{7,8,9...,17,18\}$ 27
2.6 The Battle for Wesnoth 28

3.1 Comparison of hexagonal tile patches of size 1x1, 2x2 and 3x3 . 39
3.2 Involved tiles in the one second-level differential quotient. 43

4.1 Distribution of fitness values for the assigned mutation chance values. 59
4.2 Distribution of fitness values for the assigned population size values. 60
4.3 Distribution of fitness values for the assigned mating size values. 61
4.4 Distribution of median fitness values for the assigned weighted sum value sets. 62
4.5 P-values of all parameter set combinations for the Mann-Whitney U test. 64
4.6 Fitness values and average score over all turns. 66
4.7 Minimum, average and maximum fitness values over all turns. . 67
4.8 Fitness values and average score over all generations. 68
4.9 Distribution of horizon-time ratio values over all turns. 69

4.10 Time and average time frame over all turns. 70
4.11 Village count and total number of villages for both sides over all turns. 71
4.12 Village difference and average difference over all turns. 72

List of Tables

3.1 Height value ranges and corresponding terrain representations. . 38

4.1 Hyperparameters and their respective assigned values. 55
4.2 Median and standard deviations for the fitness values of each hyperparameter. 63
4.3 Mann-Whitney U test tournament results. 65

List of Acronyms

EA	Evolutionary Algorithm
PCG	Procedural Content Generation
SbPCG	Search-based Procedural Content Generation
RHEA	Rolling Horizon Evolutionary Algorithm
CGRHEA	Content Generation Rolling Horizon Evolutionary Algorithm

1 Introduction

1.1 Motivation

Procedural generation represents a generic term for computing methods that can be summarized as the automated process of media content creation. As a branch of media synthesis this can apply to all forms of media and data such as landscapes, 3D objects, textures, meshes, models, character designs, animations, or non-player character dialogue and even more. Therefore, the area of application for procedural generation extends from mathematics, computer graphics, modeling to animation, video games and electronic music. The algorithmic generation of digital content opens up a theoretically infinite, unique set of design possibilities and results with less expenditure compared to the effort required to manually design all these possibilities by humans.

In reality, the creation of handcrafted media can never be automated or replaced by the creative process of human designers. But algorithmic approaches to this problem are definitely able to find reasonable solutions in less time and claim less maintenance cost for the development process. Therefore, the main benefit of procedural generation is the possible removal of human artists or designers in the content creation process to save time and resources [66]. This work focuses on Procedural Content Generation in Games, or often just abbreviated as PCG. The context of game content generation necessitates consideration of the design, mechanics, and constraints of the game itself [61]. This is a distinguishing feature from other types of procedural generation, such as in computer graphics or generative art, which do not necessarily have to consider constraints from external systems.

By using Procedural Content Generation in Games, video games theoretically can be shipped with an integrated designer. As the computer can run the content generation simultaneous to the game itself, there is no more necessity for a game to end as more and more content can be generated on the run to be explored by the player. The game itself will never be fully explored. This of course calls for high quality generated content and therefore a very expressive generation algorithm. Because the content has to keep up with the ever-growing desire for more interesting content to appear to the player, because the player can loose interest or get bored if he has seen one type of content multiple times already. This poses quite a challenge for the design of generation algorithms especially due to limitations of expressions via game assets as they are usually a limited set or most of them have a special context in which they have to be used. But procedural generation is also possible for game assets itself, so this challenge can be tackled as well but causes even more complex problems as this is even lower level content than a game map for example and would have to be generated on the fly as well.

Running a Procedural Content Generation algorithm along the game itself, enables access to the current game state and therefore reaction to the gameplay and not only to the state of the game before the start. This opens up the possibility to consider the needs and desires of a player from a game state aspect during the game being played. The generation then can focus on new critical aspects for video games: fairness for competitive rounds, fun for casual entertaining rounds or keeping up interest when exploring. An important step towards these critical aspects is the incorporation of the fact that a game's map is not just a simple visual representation, but also an important game component that greatly influences gameplay, outcome and difficulty. The most beautiful game world cannot entertain a player if there is no challenge throughout the game. Therefore, map generation plays a crucial role in a game's difficulty design process. Taking this a step further, a continuously generated map as the game is played can dynamically adjust the game's difficulty level and make the game more exciting or fair. This would enable reactive game content creation based not only on preferences set prior to the game being played, but also on the previous actions performed by players or AI. As mentioned, this requires online generation of the game content, optimized based on the previous game states and executed in a reasonable time so that the gameplay is not interrupted for the content generation. Procedural Content Generation (PCG) offers a search-based approach that uses Evolutionary Algorithms (EAs) to search for an optimal map, but this is usually done offline before the game is played because the process of searching for the optimal map layout is time-consuming. In the field of EAs, the Rolling Horizon evolutionary algorithm Rolling Horizon Evolutionary Algorithm (RHEA) is an online optimization approach, which in its original form serves to evolve action sequences with time constraint at each game tick [8]. This work aims to address the problem of continuously generating a map that dynamically adjusts by presenting a search-based online generation algorithm that uses a modified RHEA approach to evolve a short map sequence presented to the player as he progresses through the world.

1.2 Research Questions

The main research question of this thesis is whether a Rolling Horizon Evolutionary Algorithm can be used to generate a playable, believable and fair map for a turn-based game. In search of this answer we design, implement and evaluate a map generation algorithm for the strategy game The Battle for Wesnoth [58] that combines Search-based Procedural Content Generation with a Rolling Horizon Evolutionary Algorithm. Consequently, this thesis to answer the following research questions that arise along with the central research question:

- Can the map be generated within a fixed amount of time, which is acceptable for players?
- Can this online generation be used for dynamic difficulty adjustment during the game?
- Is the algorithm able to generate playable maps?
- Is the algorithm able to generate believable maps?
- Is the algorithm able to generate fair maps?

1.3 Thesis Structure

The next chapter, 2, covers three main background topics: Procedural Content Generation, Evolutionary Algorithms and Tile-based Games used in the context of this thesis. The Procedural Content Generation section provides basic theoretical background and typical content generation approaches. The Evolutionary Algorithms section gives insight into different accomplishments in the field of EAs, describes Search-based Procedural Content Generation in detail and then focuses on the Rolling Horizon Evolutionary Algorithm specifically. The last section covers tile-based games, hexagonal grids, and *The Battle for Wesnoth* game used in our implementation. Chapter 3 covers the general concept, structure and detailed architecture of our proposed algorithm. In Chapter 4, we report the setup, metrics and scenarios for our experiments. Additionally, we present and evaluate the experiment results and discuss them in context with the previous chapter. The paper concludes with a summary and discussion about future work and in Chapter 5.

2 Background and State of the Art

This chapter provides an overview of Procedural Content Generation, Evolutionary Algorithms, and tile-based games. Each section provides an insight into the fundamental principles and state of the art of each research area, focusing on the topics relevant to this work.

2.1 Procedural Content Generation

Procedural content generation addresses the problem of generating game content using a formal algorithm with limited or indirect user input. More generally, the abstract goal of PCG is to recreate human creativity in the context of game design [52]. These abstract definitions and views on PCG result from the fact that game content can be found in a wide variety of representations, which are associated with a wide variety of requirements and open up many possible solutions. Implementations are subject to almost no restrictions in terms of algorithmic generation, and accordingly many approaches from a wide variety of scientific fields are united under the term Procedural Content Generation. The term game content usually refers to representations of the game world such as maps, levels or scenarios, any form of interactive or collectible objects called items, and any form of textual narration such as stories, quests or characters. But game content encompasses almost anything a game can contain, including textures, music, effects or animations, and even extends to game rules that define the principles and mechanics for the game itself. Additionally, generating behavioral policies for agents can be viewed as part of Procedural Content Generation in a broader context [33].

To differentiate PCG from other research areas of general procedural generation and to clarify the context of generated game content, we use the Procedural Content Generation Wiki's definition [47] "Procedural Content Generation (PCG) is the programmatic generation of game content using a random or pseudo-random process that results in an unpredictable range of possible game play spaces." This limitation of PCG to ranges of possible game spaces allows us to consider only video game content that affects the gameplay in a meaningful way and will further on only be considered in this thesis context.

The problem of algorithmic game content generation encompasses a wide field of problems that can require very distinct solutions that can end up being strongly contrasting from each other. A list of desirable properties is introduced by Togelius et al. [61] to provide a common basis for discussing the characteristics of Procedural Content Generation:

Speed [61, p.6] "Requirements for speed vary wildly, from a maximum generation time of milliseconds to months, depending on (amongst other things) whether the content generation is done during gameplay or during development of the game."

Reliability [61, p.6] "Some generators shoot from the hip, whereas others are capable of guaranteeing that the content they generate satisfies some given quality criteria. This is more important for some types of content than others, for example a dungeon with no exit or entrance is a catastrophic failure, whereas a flower that looks a bit weird just looks a bit weird without this necessarily breaking the game."

Controllability [61, p.7] "There is frequently a need for content generators to be controllable in some sense, so that a human user or an algorithm (such as a player-adaptive mechanism) can specify some aspects of the content to be generated. There are many possible dimensions of control, e.g. one might ask for a smooth oblong rock, a car that can take sharp bends and has multiple colours, a level that induces a sense of mystery and rewards perfectionists, or a small ruleset where chance plays no part."

Expressivity and Diversity [61, p.7] "There is often a need to generate a diverse set of content, to avoid the content looking like it's all minor variations on a tired theme. At an extreme of non-expressivity, consider a level "generator" that always outputs the same level but randomly changes the colour of a single stone in the middle of the level; at the other extreme, consider a "level" generator that assembles components completely randomly, yielding senseless and unplayable levels. Measuring expressivity is a non-trivial topic in its own right, and designing level generators that generate diverse content without compromising on quality is even less trivial."

Creativity and Believability [61, p.7] "In most cases, we would like our content not to look like it has been designed by a procedural content generator. There is a number of ways in which generated content can look generated as opposed to human-created."

When considering these desirable properties for PCG, it becomes clear that these properties are usually not metrically measurable in a meaningful way. For example, speed is highly dependent on the application context, since online generation needs to be completed faster than offline generation. Other categories like creativity or reliability have no concrete definition at all that can be expressed metrically. It should also be mentioned that most of the time they cannot all be satisfied at the same time, especially not by a single algorithm. Speed, for example, is a requirement that is at odds with creativity, since more creative results typically require more computation or search time during generation. Therefore, a reasonable trade-off between these properties is required to get worthwhile results from the creation process [61].

Multi-objective Optimization Game content creation often involves more complex problems that typically cannot be described as a basic optimization problem with a single objective to optimize. In these scenarios with more than one objective, called multi-objective optimization, the overall task is to choose from a set of solutions that meet multiple objectives to varying degrees [24]. Furthermore, multi-objective problems are often non-trivial, meaning that due to the contradictory nature of the objectives, no single solution can optimize every objectives. Such is the case with desirable properties for Procedural Content Generation. Because of this trade-off characteristic or contradictory nature among the objectives, there may not exist a single solution that minimizes (or maximizes) every objective within the feasible range [57]. Instead, multi-objective optimization algorithms seek to determine the Pareto optimal solution set, a set of solutions that are non-dominated with respect to each other, meaning that no improvement in one objective can be achieved without deterioration in others [22].

Optimization problems are closely related to another type of computation problems, search problems. Without loss of generality, a search problem is a problem in which, for a given input, the best possible solution x^* is sought that satisfies all given constraints. In other words, this sometimes is associated with finding the root of a given root-finding function $g(x)$, where $g(x^*) = 0$. Deviating from this terminology, an optimization problem does not look for the best solution itself, but to optimize $g(x)$. Optimizing for the minimum possible value presents a minimization problem, but multiplying the value of $g(x)$ by -1 correspondingly transform any minimization into a maximization problem. This thesis focuses on minimization and indirectly always includes the equivalent maximization case as well when mentioning minimization. Assuming that $g(x)$ is a continuous function and the best solution found by an optimization algorithm is an epsilon-optimal solution deviating by an arbitrary small margin, we have an approximate solution of the search problem. In the case of minimizing until the smallest possible value of $g(x) = 0$ is reached, the optimization solves the search problem. Therefore, in certain aspects, search and optimization problems are effectively equivalent [55]. In general, search problems often deal with systems of equations and inequalities that represent a non-differentiable class of functions, and hence direct search algorithms do not have access to $g(x)$. However, accessing $g(x)$ directly is not mandatory, since a black-box optimization is sufficient to find the best solutions or a solution with a certain minimum quality needs to be found.

2.1.1 Categories

Procedural Content Generation algorithms can be categorized in many ways and subdivisions. The following section presents the most common approaches in this area, categorized according to the underlying main principle of generation.

Search-based The search-based approach to PCG relies on Evolutionary Algorithms to solve the search problem of finding meaningful content representations in the search space of all possible content representations. Inspired by natural Darwinian evolution, a population of solutions is created that is modified by selection and mating operators over several generations until a stopping criterion for the search process is met. EAs and Search-based Procedural Content Generation are explained in detail in their own separate subsequent section 2.2.

Constraint-based Using logic programming, the constraint-based approach describes the problem to solve through constraints as opposed to describing how to solve the problem. The constraints are then passed to a constraint solver such as in Answer Set Programming [51], which searches for solutions that satisfy the problem described. Solvers typically represent a kind of black-box optimizer, but are highly implementation-dependent and can vary greatly in their functionality. In general, the search process consists of assigning each variable a value in its range, so that in the end all the constraints are satisfied. Therefore, the result is most often a single solution, but could also be a list of feasible solutions that meet the desired criteria, which would require a further decision-making process to choose the best one.

The constraints can allow faster convergence time to find solutions compared to the search-based approach because they reduce the total search space of the problem. However, suitable constraints require a great deal of knowledge about the game, since otherwise the search space could be reduced in such a way that important solutions are left out or invalid solutions are found by omitting important constraints. Therefore, the computation time is strongly affected by the quality of the constraints, but also has the potential to rapidly generate levels that can meet the real-time specifications of human designers [53].

Rewriting Systems Rewriting systems are a wide range of methods that replace partial terms of a formula with other terms derived from the research area of theoretical computer science. In a procedural generation context, they enable deterministic or non-deterministic content creation, starting at a given point and applying replacement rules.

One of the most popular rewriting systems is the L-system [30], inspired by Chomsky's [4] work on formal grammars and biological processes in cells. While formal grammars apply replacement rules sequentially, L-systems work in parallel and can even be extended to allow the generation of full road networks [38] or procedural geometric modeling of complex 3D models [35].

They are also most commonly used to generate fractals. Fractals describe a wide range of geometric shapes that resemble naturally occurring branching patterns seen in plants or other cellular structures. While they appear to mimic natural processes such as plant growth or erosion, their generative nature allows them to contain detailed structure at arbitrarily small scales.

This is a desirable property in Procedural Content Generation, since imitations of naturally generated real-life structures appear to represent high quality generated content. Additionally, games typically present content at different scales, such as an overview of the entire world map, which is then split into multiple low-scale levels based on the higher-level map. Fractals enable the generation of exactly such content and find their most suitable use case in these scenarios.

Constructive Approaches Similar to rewriting systems, constructive approaches also start at a given point, but then gradually generate content instead of replacing it. This means that only one solution is generated per run, typically via cellular automata or similar systems. A cellular automaton is a discrete computational model consisting of an n-dimensional grid with a number of cells in a finite state and a set of transition rules. By applying the transition rules to all cells simultaneously, a new generation is created as each cell is transitioned to a new state based on its own current state and the state of all cells in its neighborhood. Hence, the neighborhood defines which cells around a given cell will affect its future state.

The best-known example is Conway's Game of Life [13], a two-dimensional zero-player game that combines very simple programming with resulting emergent and self-organized behavior. Cellular automata are widely used to model environmental systems such as fluid flow, fire, rain or explosions in games and are also capable of generating procedural terrain [18].

Noise Functions [25] "A noise is a stationary and normal random process. Control of the power spectrum is provided, either directly, or through the summation of a number of independent scaled instances of (typically band-limited) noise".

Commonly understood in physics as a disturbance variable with a broad, non-specific frequency spectrum, noise has applications in computer graphics and PCG as a random and unstructured pattern that can be useful for pattern generation or efficiently adding a source of rich detail to synthetic images. Mathematically, noise can be understood as a multivariate random variable generated by various stochastic processes. White noise, for example, can be viewed as a combination of statistically independent random variables, each component having a probability distribution with zero mean and finite variance.

In practice, noise is represented as a series of random numbers arranged in an n-dimensional grid, typically as two- or three-dimensional matrices of real numbers. These numbers can then represent the brightness of associated pixels or the elevation of a specific point in a height map. The research area of computer graphics uses noise for procedural texturing to model complex materials and objects such as terrains or shapes and can even be extended to the animation of water surfaces, for example. Inspired by the observation that natural systems appear noisy, the basic idea of creating natural lookalikes through procedural generation is to add noise to the content. However, noise usually has to meet some properties, the most important of which is smoothness, which means that local changes in noise should be gradual, while global changes can be larger.

Perlin proposed the first known implementation of a gradient noise function in 1985 [42]. The basic implementation involves three steps: creating a vector grid and choosing pseudo-random gradient directions at the vertices, computing the dot product between the gradient vectors and their offsets, and smoothly blending between them to achieve smooth transitions. Typically, multiple layers, called octaves, are combined in addition to create a better fractal appearance in the end. Later, Perlin addressed problems of his original noise function and designed an algorithm to reduce the computational complexity of scaling to higher dimensions and reduce the visible lattice artifacts, called simplex noise [43]. Implemented by default in most computer graphics software packages today, procedural noise is still widely used, particularly in video games, and continues to be researched [21], [2], [26].

2.2 Evolutionary Algorithms

Evolutionary algorithms are a class of population-based search algorithms that belong to the stochastic, metaheuristic optimization methods. Fundamentally, metaheuristics represent general computing techniques that are designed to solve numerical and combinatorial optimization problems and provide a sufficiently good solutions over several iterations [24]. The specific population and search-based aspects of EAs are inspired by the Darwinian principle of evolution to create simulated evolutionary optimization algorithms. The general assumption is that biological evolution is capable of solving challenging adaptation problems and providing complex solutions represented as life forms, and therefore provides a working rationale for addressing optimization problems. For a given optimization problem, the resulting algorithms apply evolutionary principles such as mutation, selection, and reproduction to a population of candidate solutions to produce offspring and thereby increase variation in the population. Another selection operator, typically survival of the fittest, reduces the population back to its original size, reducing variation by dropping the least fit individuals.

Based on the theoretical definition of optimization problems, the goal is to find the single best solution from all feasible solutions, but as mentioned in Chapter 2.1 on multi-objective optimization, it is more reasonable to find the set of the n-best solutions for the given problem. For these reasons, in the further course of this work we always consider the general goal of Evolutionary Algorithms to find the n-best solutions instead of the single best solution. Optimization problems almost always have one or more constraints, a condition that the solutions must satisfy. For this reason, among these n-best solutions, there may be non-valid solutions that are less in conflict with the constraints than others of higher quality. Constraints serve as a measure of feasibility, and not conflicting with constraints means preserving feasibility, since feasible parents produce feasible offspring, which is a desirable trait. In order to find these n-best solutions, the most promising intermediate candidates are successively improved over several generations, using the evolutionary principles already mentioned. The incorporation of previous knowledge about the optimization problems allows for the design of problem-specific evolutionary operators and the integration of these in the search process to cope with the varying challenges of complex problems [1]. The aforementioned evolution process and its structure can be better observed in Figure 2.1, which shows a graphical symbolic representation of a generic evolution algorithm.

The search process begins with the typically arbitrary initialization of the population of individuals, each individual representing a search point in the space of possible solutions to the given optimization problem. The generation cycle is then started by decoding this population of search space points into their solution space representation so that they can be evaluated in the environment, the solution space of the problem. This results in a quality information, the so-called fitness value, being assigned to each search points to indicate a ranking among the existing solutions. After each evaluation, the terminal con-

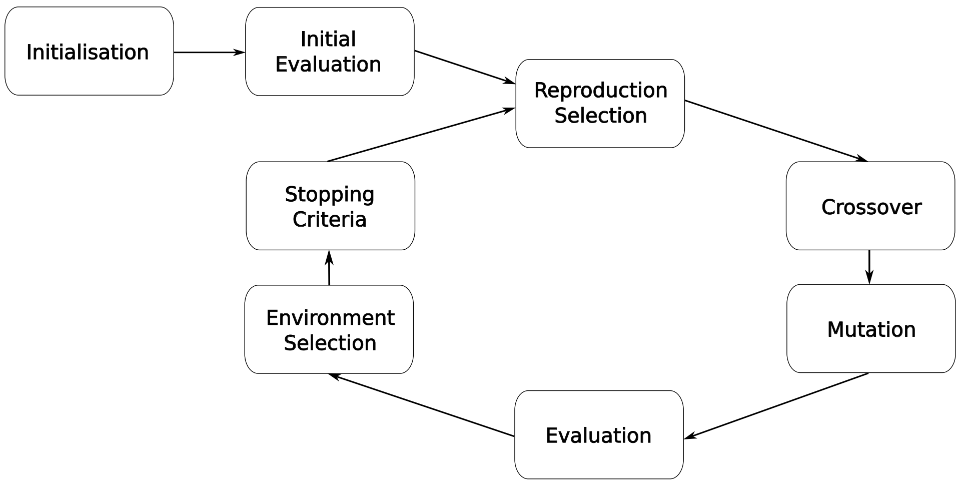


Figure 2.1: Visualization of the basic process of Evolutionary Algorithms.

dition is checked to verify if the optimal or sufficiently good n-best solutions have been found. If that is not the case, the cycle continues by selecting favorable individuals to reproduce based on their fitness. These selected individuals will create new solutions based on the recombination of their own search space coding. These new individuals are additionally mutated, modified on a small scale, to introduce independent innovations.

These new individuals are then integrated into the population and evaluated to assign them a fitness value. From this larger population, individuals are selected and sorted out in order to keep the population size constant over the generation cycles. Since a growing population means longer running times, it is generally not considered favorable to keep more potentially weaker solutions at the expense of increased computation time. Afterwards, the generation cycle is run through again until a termination condition is met after the fitness assessment of the population. By recombination of favorable solutions and only keeping candidates of reasonable quality in each generation cycle, the algorithm eventually evolves the population into more desirable regions of the search space to find the optimal n-best solutions [3]. Algorithm 1 shows the general scheme of an EA and serves as a computational representation of the common basic functionalities of Evolutionary Algorithms.

Algorithm 1 General Scheme of an Evolutionary Algorithm by [24]

```
procedure evoalg;
begin
    $t \leftarrow 0$;                                  (*initialize the generation counter*)
    initialize pop(t);                                 (*create the initial population*)
    evaluate pop(t);                                   (*and evaluate it (compute fitness)*)
    while not termination criterion do                 (*loop until termination*)
    $t \leftarrow t + 1$;                              (*count the created generation*)
    select pop(t) from pop(t - 1);                     (*select individuals based on fitness*)
    alter pop(t);                                      (*apply genetic operators*)
    evaluate pop(t);                                   (*evaluate the new population*)
    environmental selection (pop(t), pop(t - 1));
    (* select individuals for the next population*)
    end
end
```

2.2.1 Building Blocks of Evolutionary Algorithms

The following section provides a brief depiction of the basic underlying mechanisms in the evolutionary search process, focusing on those aspects that are most relevant to the concepts and ideas discussed in this work.

Encoding Again following natural inspiration, individuals are encoded via chromosomes, which represent a sequence of computational objects such as numbers, bits, or characters. For the sake of simplicity, in Evolutionary Algorithms, individuals usually have only one chromosome, but with multiple variables or computational objects represented by genes. Each variable has its own variable domain expressed as possible alleles. The entire genetic representation of an individual is referred to as a genotype or encoding and defines the overall dimensions of the search space, the set of all possible genomes. However, in order to evaluate all individuals in the population for the given optimization problem, these variables must be mapped to implementable solutions. These instances of the optimization problem lie in the solution space, the environment of the problem. In the so-called genotype-phenotype mapping, encoded individuals are transformed into their decoded representation in the solution space, the phenotype. This mapping, which connects search and solution space, is typically performed with a decoding function.

If the encoding is too large or too complex, the search space becomes correspondingly high-dimensional, which complicates the search process for the EA. Optimal solutions are generally harder to find in larger search spaces, reducing the overall effectiveness of the search as a whole [62], known as the "curse of dimensionality". The genetic representation not only affects the efficiency of the search, but also biases the search process towards different parts of the search space. Even though individuals can be evolved to satisfy the same evaluation function and reach similar fitness, the results mapped to the solution space can look very different. A desirable property for the encoding is locality, which describes that similar genetic representations should result in similar phenotypic proximity. Altering one gene on the chromosome should not lead to a completely different solution.

Initial Population Mathematically, an Evolutionary Algorithm population is a multiset of candidates because it is technically possible for identical solutions to exist due to the expression limitation in the chromosome representation. Usually all individuals are generated as a random chromosome, but it is possible to choose other methods if constraints on the representation of individuals have to be satisfied or if the random initialization generates invalid candidates that have to be discarded or repaired.

Fitness The fitness value of an individual is a measure of the quality of the performance of this solution in the problem to be optimized. In most cases, the function to be optimized and the fitness function are identical, or at least the optimization problem provides a fitness function with which solution candidates are to be evaluated. But similar to the remarks on population initialization, constraints can additionally be incorporated in the fitness function if the problem requires these to be satisfied in order for a solution to be accepted. In addition, the fitness function can be used in conjunction with constraints to introduce a bias toward certain additional desirable properties of a solution.

This inclusion can also be expressed via multiple fitness functions, each of which provides a measure of quality related to different aspects of performance in the optimization problem. However, the classical selection operators require a single value to rank the members of the population, which is why it is common to design a combination of several fitness functions into one final result. A most straightforward approach is to use a weighted sum across all fitness functions, but this requires careful tuning of these weights to account for the interaction of different fitness values, as they may depend or even counteract each other.

Selection Selection is one of the three main genetic operators guiding the Evolutionary Algorithm in its search for an optimal solution to a given optimization problem. Based on the assumption that better performing individuals produce offspring of higher quality by passing their own strong genes to their children, selection operators should favor the highest ranked individuals for reproduction based on fitness. Selection operators are an essential but completely independent part of the EA, since they operate on individuals and their fitness regardless of the structure of the search space. The degree of impact that fitness has on selection is called selective pressure. Selection should always be proportional to fitness due to the aforementioned assumption and the fact that selection without selective pressure, meaning without regard for fitness, essentially degrades the search process to a random search that is unlikely to find an optimal solution in reasonable time. On the other hand, directly ranking and selecting individuals based on their fitness can lead to a domination problem. In this scenario, all other individuals are suppressed by a single individual with a very high quality that will almost always be selected.

A specific selection operator to address the domination problem and control selection pressure is tournament selection. Rather than rank and select individuals in direct proportion to their fitness, individuals must first win in a tournament to be selected for reproduction. A tournament is a competition between k individuals, uniformly drawn at random, with the winner then being determined based on fitness. Therefore, the solution with the best fitness of those k competitors is selected and all contestants are then returned to the draw pool for the next tournament. With this method, fitness only indirectly contributes to reproductive success, since all individuals have an equal chance of participating in a tournament and only the chance of winning a tournament is determined by fitness.

The parameter $k \in 2, 3, ..., |pop|$, which represents the tournament size, is a selective pressure control where larger tournaments also apply greater selective pressure. This is exemplified by the example of the largest tournament with size $k = |pop|$, which degenerates into direct fitness rank selection, selecting only the best individual each time. Despite the obvious fact that the individuals with the highest fitness are the most likely to win a tournament, it is not impossible for worse individuals to reproduce given the possibility of being drafted into a tournament where all other contestants have a lower fitness score than they possess. Especially with a small tournament size k, lower quality individuals are more likely to encounter this scenario in the selection process. The only individuals without any chance to win a tournament are the worst $k-1$ of the population. Conversely, larger tournaments increase the chance of including one of the best individuals, nullifying less fit participants' chances of winning in the tournament.

The previous discussion of selection only focused on the selection of individuals for reproduction, but once these solutions produce offspring, the population grows and can noticeably slow computation time if the population simply keeps growing with each generation. Here, environmental selection addresses this issue by deciding which individuals to drop after a generation and which individuals to advance to the next generation. Most commonly, the combined population of existing individuals, their number is denoted by μ, and the newly created offspring, denoted λ, is considered and exactly μ individuals are selected for the next generation cycle. This approach has the advantage that high quality solutions can last for generations, even if they may not produce better offspring through reproduction, and is known as elitism. Elitism ensures that the fitness already achieved by individuals through the search process does not decrease from one generation to the next and leads to better convergence properties in which local optima are consistently approached [23]. Converging to local optima too fast and too early opens up the discussion about the usefulness of this approach with this disadvantage, but it has the advantage that the quality of the n-best individuals never deteriorates over time.

Mutation and Crossover Besides selection, mutation and crossover are the other two main genetic operators. Their main purpose is to modify and recombine chromosomes to produce new candidate solutions that are similar to their parents. A Crossover operator involves more than one parent solution, usually two, and recombines their genes to produce offspring. Since the selection operator generally favors and selects the most suitable individuals for reproduction, the crossover operator then shuffles the genetic information of these parents in the expectation of creating an even better solution. Subsequently, the mutation operator introduces more genetic diversity into the generated offspring through small variations of their genes. The variations of a child solution are independent of their parents to prevent the pool of individuals from becoming too similar over the course of evolution, and can be viewed as introducing innovations into the gene pool.

As with other building blocks, it should be clear that the use of a reproduction operator is only a general requirement for a working EA. Depending on the problem and the chosen encoding, the genetic operators can be very generic or highly problem-specific, and are usually chosen to closely match the individual's chromosomal representation. The most well-known crossover operator is the one-point crossover, in which a random intersection point is determined and gene sequences on one side of the intersection are swapped between the parent chromosomes. The two-point crossover works on the same principle, but selects two intersections and swaps gene sequences between the two intersections. This can be generalized to a variable number of crossover points with the n-point crossover, in which there is an alternating swapping and non-swapping of gene sequences between two consecutive intersections. The classic example to elucidate a mutation operator involves individuals with their chromosome encoded as a bit string, meaning that each gene in the sequence can be expressed as either zero or one. The simplest operator, bit mutation, then randomly flips alleles by turning a zero into a one and vice versa.

Termination Conditions The search process usually takes a relatively long time, but should never take forever, which is why a termination condition is essential for any EA. The overall goal is to find the final optimal solutions, but the optimal points of the problem are not known beforehand, so other intermediate measures are used to prevent an unending search scenario. Typical termination conditions involve a predefined quality that an individual must achieve, a certain number of generation cycles that have been completed, a measure of whether there is still visible progress in terms of fitness performance, or a combination of these.

Hyperparameters Only touched on in this subsection, but still important to mention: For a complete specification of an Evolutionary Algorithm, several hyperparameters must be defined before starting the search process, such as the population size μ, the number of offspring to be produced λ and all other parameters required by the genetic operators. The search for the optimal set of hyperparameters is a scientific research topic in itself [19], [7], but even with less complex problems these cannot simply be assigned arbitrarily, but should be determined with care and taking into account the problem specification.

2.2.2 Rolling Horizon Evolutionary Algorithms

Rolling Horizon Evolutionary Algorithms (RHEAs) are a subclass of Evolutionary Algorithms that focus on planning and policy generation for game agents under real-time constraints. Originally introduced by Perez et al. [29] as a rival method to Monte Carlo tree search algorithms, a popular heuristic search method for real-time decision making, they have become a similarly popular and researched method in the context of General Video Game Playing [28].

The obvious approach to addressing the problem of decision making in games with EAs would be to pre-search for the best possible actions in an offline model of the game and then train an agent to perform those actions during gameplay. This is a relatively inflexible approach that does not allow dynamic adjustment or reaction to new in-game situations, since the search process is already complete. RHEAs integrate the search problem into the game itself by evolving an action sequence for a short finite period of time, typically a few milliseconds, online during the game in an internal model where actions can be simulated and evaluated [8]. Then the best action found is carried out, the change in the game state is observed and a new, short-lived search process is started in order to find the subsequent best action for the new game state. This process is repeated until the game is over.

The genetic representation of a Rolling Horizon Evolutionary Algorithm depends on the possible action space of the game, but is always mapped to the phenotype representation as an action sequence from an initial state to the last considered action. Similar to EAs, the fitness function for evaluating actions is kept general to avoid expressive limitations and places a general focus on performing adequately to win the game overall. The termination criteria is not tied to reaching a fixed number of performance score or even finding the most optimal solution, but limited based on the search algorithm's fixed exploration range, called horizon, a limited look-ahead in the form of a time frame, but in much smaller dimensions than the typical search process would take.

RHEAs have become a core component of General Video Game Playing framework [9] and are constantly improved and modified in order to continuously improve their performance further [10], [11]. The general approach allows for the application to a varying kind of games as they are designed with the goal of general video game playing in mind [31], [49].

2.2.3 Search-based Procedural Content Generation

Search-based Procedural Content Generation covers Procedural Content Generation algorithms that use EAs or other stochastic optimisation algorithms to search for and generate good game content. For a detailed explanation and discussion of Evolutionary Algorithms, we refer the interested reader to Section 2.2. The term Search-based Procedural Content Generation is not limited to EAs, but allows all forms of heuristic and stochastic optimization algorithms [63]. However, within the context of this work, we mainly discuss the most commonly used metaheuristic for SbPCG, Evolutionary Algorithms.

Mutation and crossover operators, essential components of an EA, are usually chosen highly variably, since they always have to match the problem-specific encoding. As the generation of game content covers a wide range of problems and thus a large number of problem representations, the operators also vary greatly. Unfortunately, little attention is paid in the scientific literature to the specific implementation of these operators [59],[17],[12],[48].

The scholarly works have agreed on two central problems that have to be addressed in the construction process of such an algorithm: the content representation, which defines the search space of the problem, and the evaluation function, to determine the quality of solutions [60]. Only a combination of appropriate content presentation and a meaningful evaluation function enables the evolutionary search to find interesting, diverse content in a reasonable amount of time. These two central problems are discussed in more detail below.

Content Representation Since solutions in Evolutionary Algorithms are encoded as a sequence of computational objects, they can take any form depending on the required application and the presupposed problem. In the context of game context generation, the genotype-phenotype mapping typically contains instructions for creating game content such as a level, maze, or map, and the phenotype involves the visual representation of the actual content to be generated for the game. Choosing the right encoding is of paramount importance as video games tend to have complex data structures for their content, i.e. a graphical model represented as a mesh of triangles, or a map that spans thousands of tiles, or the whole story as a text spanning several chapters [60]. Because the genetic representation of individuals is directly correlated with the dimensionality of the search space, the chosen genotype influences the efficiency of the search process and the vastness of content that the algorithm will be able to cover.

Evaluation Function The fitness function assigns a fitness score to each solution as an indication of its quality and is the main guide in the search process towards better solutions in the search space. Therefore, a badly designed fitness function can prevent the entire evolutionary process from working as intended and prevent high-quality solutions from being found. Crafting an evaluation scoring function to map desirable content quality in a video game context to a numeric value is an ambiguous task. This can depend heavily on the desired type of content and its in-game functionality to be output [63]. The most popular example of this is the term fun, as this is a commonly understood concept, but one that seems impossible to define and articulate without leaving room for misinterpretation. For example, some players may describe a challenging, competitive round as entertaining, while others prefer more peaceful rounds that require less consideration and planning to achieve a win in order to have fun.

In general, fitness functions are not subject to strong design restrictions and one could come up with an almost infinite number of them, since they can also be designed specifically for the problem. However, Togelius et al. [63] distinguish between three key classes of evaluation functions in the context of SbPCG, which are summarized below.

Direct Evaluation Functions evaluate content based on the content's phenotypic representation in the game. As is customary in evolutionary algorithm, solutions are decoded into the solution space and evaluated directly in the problem-specific environment. Therefore, characteristics of the generated content directly correlate with the associated quality. These fitness calculations have the advantage of being fast to implement and execute, and can provide a quality score at all times, including during gameplay. The resulting disadvantage of this forward approach is that the connection between game content features and desired or associated fitness is non-obvious. Because game content can interact with the game and the player in various complex ways, the task of devising a direct fitness function is very challenging.

Simulation-based Functions evaluate content based on calculated statistics of on AI agent playing through the game. Agent behavior and the resulting game states provide information on how generated game content is to be evaluated. The most common are two types of assessment tasks that also indirectly set the requirements for the AI agent, playability and player experience. In the case of playability, the ability to reach the end of the game or get as far as possible or last as long as possible are crucial qualities that an agent has to exhibit. The subject of player experience poses a very general but also difficult task, which in short is often tackled with an AI agent capable of mimicking human behavior.

Interactive Functions evaluate content based on interaction with a human, either implicitly or explicitly collecting data from the player. Implicit data collection is based on assumptions and knowledge about the connection between player behavior and content quality. As an example, Hastings et al. [15] attributed the number of times a procedurally generated weapon was selected to the overall popularity of the content. Since this method relies on implicit knowledge and assumptions, the results are always bound to the incorporated understanding of game and design mechanics. Explicit data collection takes into account direct decisions about the quality of generated content, providing more insights and reliable information. But this usually requires a break in gameplay and a change of focus for the player. Switching from gaming immersion to evaluating gaming experience and decision making takes a longer time overall. The general requirement of having a human integrated into the process is on the one hand a challenge and another resource that needs to be managed. But on the other hand, a useful decision maker and accurate estimator of player experience for the game content design process.

These three different classes of fitness functions all have their advantages and disadvantages. Which of these functions is generally most useful for Procedural Content Generation cannot be decided in general terms. Fundamentally, the design and application of a fitness function to an optimization problem is always highly task-specific. For this reason, it can be beneficial to have more than one fitness function to capture multiple aspects of fitness to account for more than one strict definition [60]. This is consistent with the conclusions drawn in Section 2.1 on the subject of multi-objective optimization.

2.3 Tile-based Games

Tile-based video games represent their playable game world on the screen as a grid of tiles. While tiles are compact polygonal graphical entities, mostly triangles, squares, rectangles, or hexagons, the set of possible images that can be displayed by the game is called a tile set. Dating from the earlier days of video game development, when computers were limited in their computational capabilities and ability to display rich textures, running a game and displaying graphical information at the same time required careful game design. Compared to always rendering the entire frame at once, tiled rendering reduces memory, bandwidth, and processing time, making it possible to conserve these resources in times when video games, for example, had to fit into cartridges. Today, the hardware of modern computers allows for more resource-intensive graphics display, and tile rendering is only visually used as a design choice for its highly recognizable visual appearance.

From a game world-structuring and interaction perspective, hexagonal grids have clear advantages over the traditional square grid counterpart. First, neighbouring tiles always share a common edge, which means that no two cells touch at just a single corner, compared to squares. Second, all neighboring tiles in the grid are equidistant from each other because the distance from the center of one tile to the center of the six adjacent tiles is always the same. In a square grid, the distance from the center to the four diagonal neighbors is skewed by a factor of $\sqrt{2}$ compared to the distance to the adjacent vertical and horizontal neighbors. Hex grids are most often considered in strategy games due to the influence that the equidistant property and additional neighboring tile have on tactical gameplay. Circular attacks and effects like explosions work in a more natural radius, and unit movement is also more balanced compared to square grids.

Tile-based Games in Science Tile-based games are often associated with Procedural Content Generation, especially map generation, in scientific research [33],[46]. They conveniently subdivide their content presentation into smaller, manageable pieces with local characteristics. Notwithstanding this division, tiles provide enough functionality for endless possibilities in game content creation [36],[14],[54].

2.3.1 Hexagonal Grids

A hexagon is a 6-sided polygon with 6 corners and if these are regular, i.e. all sides are the same length, 120° interior angles in each corner. If the opposite corners of a regular hexagon are joined together, the inner surface can be represented by six equilateral triangles. The hexagon, along with the square and equilateral triangle, is the only equilateral polygon that allows regular tiling of a plane, that is, edge-to-edge tiling. In a plane tiled with a hexagonal grid, each hexagon is connected to its neighbors by entire edges, and never just by corners or portions of an edge. From another angle, one could say that each corner of the grid is shared by 3 hexagons and each side by 2 hexagons. The typical orientation for hexagons in a grid is either as horizontal rows shifted one below the other, or as shifted vertical columns. Figure 2.2 show a comparison between both possible hex grid orientations.

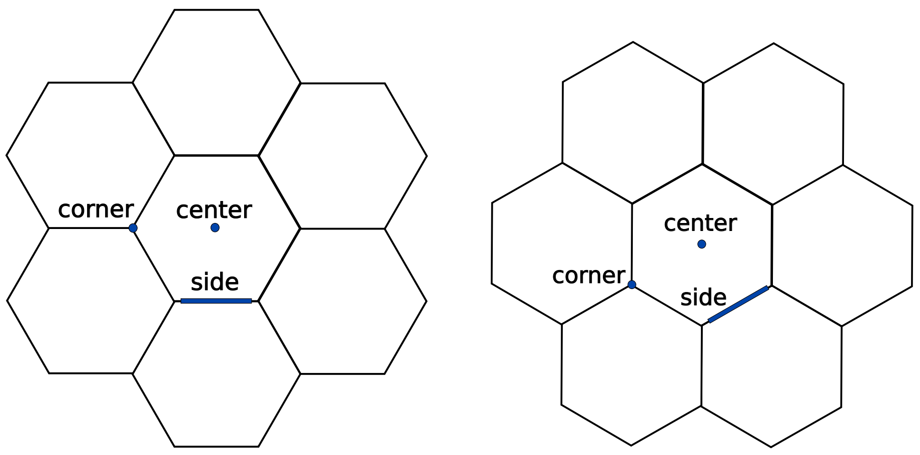
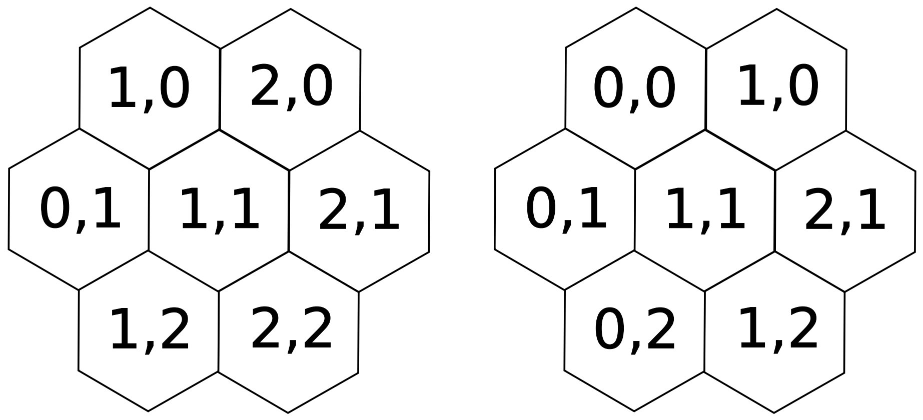


Figure 2.2: Comparison of hexagonal alignment of vertical columns and horizontal rows.

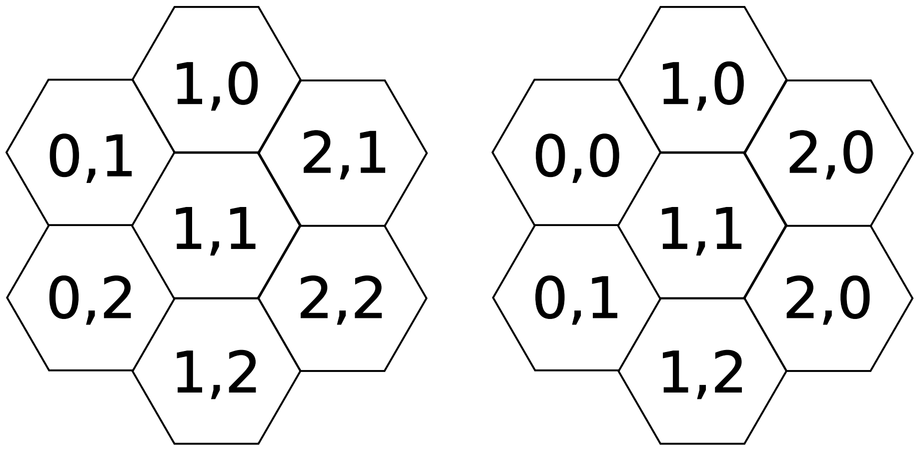
Coordinate Systems When defining a hexagonal grid, not only the orientation needs to be defined, but also how coordinates are assigned to each tile, as there are several possible approaches. On his website [41], Amit Patel presents a comprehensive compendium on this subject, which contains various approaches, common formulas and algorithms for hexagonal grids, on which the following section is based.

Offset coordinates

The most common approach is to offset every other column or row. You can either offset the odd or the even column/rows, resulting in two variants for each orientation of the hexagons in the grid. The resulting possible representations are exemplary shown in Figure 2.3.



(a) Horizontal layout with odd and even offset



(b) Vertical layout with odd and even offset

Figure 2.3: Overview of hexagonal coordinate system layouts.

Cube coordinates

Another way to look at hexagonal grid coordinates is to develop a coordinate system with three main axes X, Y and Z, as opposed to the two we have for square grids [16]. A point on the hex grid is thus described by three coordinates (x, y, z). In this system, the hexagonal grid is actually a diagonal plane that is a cut through a cube in 3D, described by $x + y + z = 0$.

Depending on the chosen orientation of the grid, there is either a horizontal axis or a vertical axis and the other two run symmetrically diagonally and cross at the coordinate origin. Visually, in the vertical representation, the X-axis refers to northeast/southwest movement on the grid, the Y-axis to northwest/southeast movement, and the Z-axis to east/west movement. The three axes are then traversed similarly to Cartesian coordinates, but moving a hex tile changes two coordinates instead of one.

Axial coordinates

Axial coordinates are an extension of the cube coordinate system by incorporating the fact that the plane defining the X, Y and Z axes in cube coordinates is constrained by $x + y + z = 0$. As can also be seen visually, points on the Z axis always satisfy $z = -x - y$ due to axial symmetry [32]. Therefore, the z-coordinate is taken into account, but not stored for the coordinates of the points, only calculated when needed. Figure 2.4 shows a comparison of cube and axial coordinates in the case of a vertical column hexagonal grid.

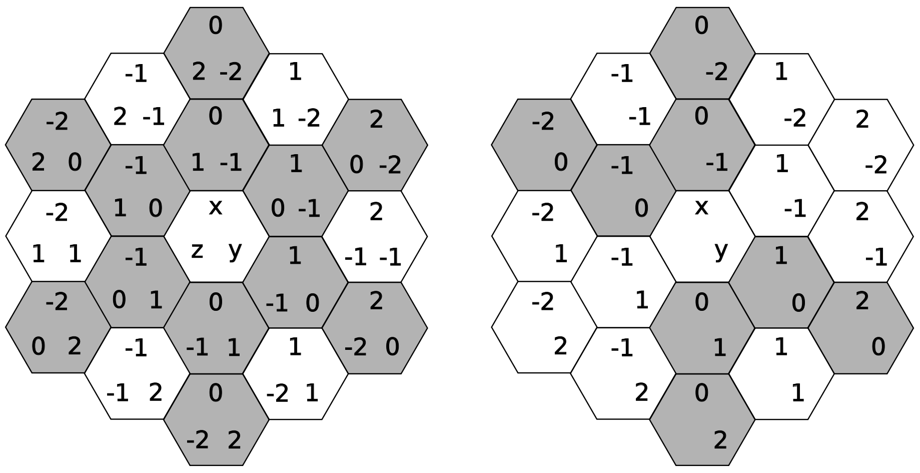


Figure 2.4: Cube and axial coordinate system.

Neighborhoods Since access to single tiles and their surrounding neighboring hexes is crucial for map generation, we formally define the neighborhood relations for the case of vertical offset coordinates as follows:

For a given hexagonal tile in the grid h the first level neighborhood N_1 contains all tiles n_i that are in reach of one step from h, meaning $N_1(h) = \{n_i | distance(n_i, h) = 1\}$. For all tiles not lying on the edge of the grid there exist six tiles in the first level neighborhood and index them starting from one to six clockwise. Similarly we define the second level neighborhood as $N_2(h) = \{n_i | distance(n_i, h) = 2\}$. Figure 2.5 visualizes these relationships again graphically. Specifically for this work, we further subdivide N_2 into two subsets, N_{2k} and N_{2j}. N_{2k} contains all tiles in the second-level neighborhood with an even index as $N_{2k}(h) = \{n_i | distance(n_i, h) = 2 \wedge i \mod 2 = 0\}$. Accordingly, $N_{2j}(h) = \{n_i | distance(n_i, h) = 2 \wedge i \mod 2 = 1\}$

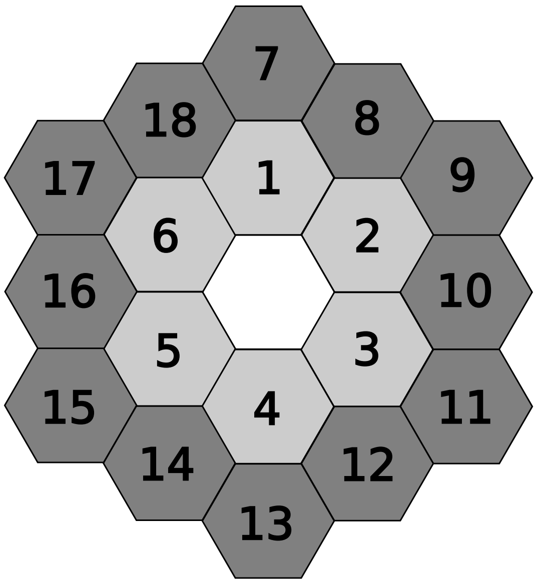


Figure 2.5: First and second level neighborhood with $N_1 = \{1,2,3,4,5,6\}$ and $N_2 = \{7,8,9...,17,18\}$

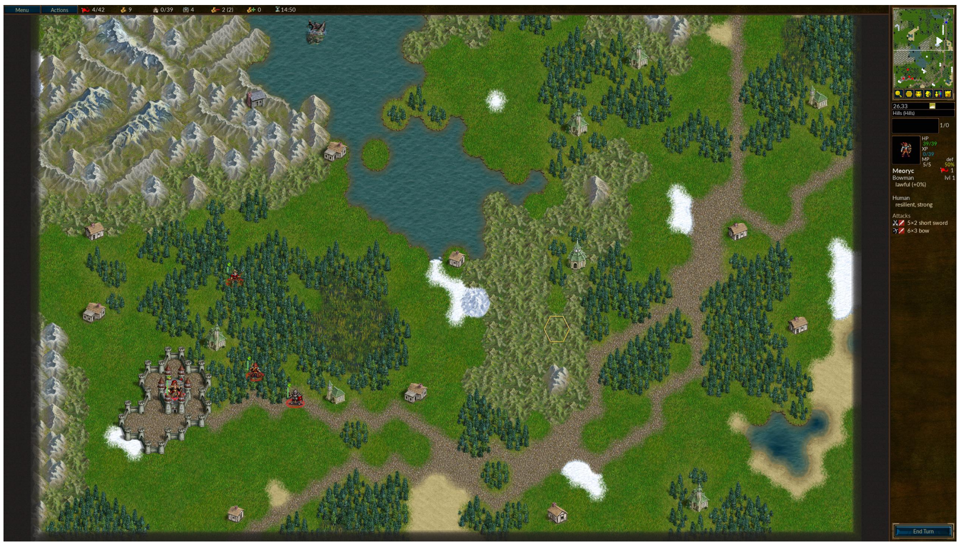


Figure 2.6: The Battle for Wesnoth

2.3.2 The Battle for Wesnoth

The Battle for Wesnoth is an open-source, turn-based, tactical, fantasy-themed strategy game playable on a variety of platforms including MS Windows, Linux, and MacOS X. A prominent representative of the turn-based strategy genre, the game mostly revolves around planning and finding the best strategic approach to resource management, unit movement, and logistics against another player.

Game Mechanics and Gameplay Two or more players play against each other in a round of The Battle for Wesnoth, with the game allowing different numbers of human and artificial intelligence players in single-player and multiplayer modes. Players are referred to as sides in the game and each side starts with a few units, one of which is a commander, a keep where units can be recruited by the commander, and some gold, the main resource to recruit units and pay for their upkeep. The main resources to fight for on the map are villages, as they generate gold per turn when captured by one side, and other keeps, as they allow for an accelerated increase in military presence at strategic points by enabling recruitment at their place.

A typical round then revolves around building an army by recruiting new units and gathering more resources while fending off the enemy forces from doing the same. Winning against enemy forces requires either more or better units, which requires more resources, but capturing and defending more resources requires more or better units over time. This creates a constant circle of demand between units and resources that keeps the game interesting and flowing as all sides try to expand their own army and supplies, leading to constant confrontations with other sides on the map and requiring strategic unit management for battles over the resource sovereignty.

The game world is presented as a tile-based hexagonal map, with each hexagon containing a string representing an overlay texture. An overlay texture is a two-layer texture that contains a base layer, the graphics rendered at the bottom of a terrain texture, e.g. the water under a bridge, and the overlay, a specific top graphic for terrain such as bridges, forests or villages. The game ships with an already implemented map editor that contains several map generation algorithms, the parameters of which can be adjusted manually. Figure 2.6 shows a screenshot of the game map with the various terrains and tiles.

Ludii Categorization The Ludemic General Game System [44], Ludii for short, is a general game system that allows for a human-understandable categorization and description of games. Originally designed to model historically traditional strategy board games, the categorization principle is universally applicable. Games are presented as structured sets of ludemes, high-level comprehensible game concepts, or units of game-related information. Below we provide a minimum description of The Battle for Wesnoth that results in a legal Ludii game description.

```
(game "The Battle for Wesnoth"
    (players 2)
    (equipment
        {(board(hex Rectangle 40)) (piece "Leader" Each)
    )
    (rules
        (play (forEach Piece))
        (end
            (if (no Pieces P1) (result P1 Loss))
            (if (no Pieces P2) (result P2 Loss))
        )
    )
)
```

The description follows the basic scenario of two players competing on a standard 40 by 40 hexagonal tile grid. First, the name of the game, the number of players and the board with its shape and size are defined. In addition, since the sides in The Battle for Wesnoth are represented by leader units, we define that each player should have a piece named "Leader". Next, the minimum required game rules, the playing and ending rules are defined. The simple playing rule applied here merely describes that Ludii loops through all the pieces of a player and extracts legal actions from the pieces to be executed. Equally basic, the ending rule is defined as a rule that comes into effect when either player has no remaining pieces and therefore loses the game. The actual players are denoted here by P1 and P2. This description forms only the most minimal principle of the game description for The Battle for Wesnoth, which generally allows for much more complex customization and has more rules. Overall, The Battle for Wesnoth is a zero-sum game as only one side can win which automatically means the other side loses and is therefore a purely competitive game.

Wesnoth Markup Language The Battle for Wesnoth features a highly modifiable engine built on their own markup language called Wesnoth Markup Language and the Lua programming language. The Wesnoth Markup Language is used in all parts of the game, from creating new content to changing the game scenarios, units, save files or even the interface layout. As a markup language, Wesnoth Markup Language files are plain, human-readable text files that contain a set of markup instructions, plus macros for extensive use that allow Wesnoth Markup Language code to be stored in a macro variable that can be called later, instead of writing the same code again. The Wesnoth Markup Language contains two basic elements in its syntax: tags and attributes, where attributes consist of keys and values. For example:

```
[tag]
    key=value
[/tag]
```

Tags partition information and serve as an identifier for any information unit, such as the definition of a side or an event when the Wesnoth Markup Language file is processed. Attributes contain data that are uniquely assigned by their prefixed key, which, in addition to the actual content, also defines the type of data. Since tags are the main level structure for the Wesnoth Markup Language, they also allow tags within another tag to enable a child-parent hierarchy. Additionally, the [lua] tag allows for custom code execution during gameplay via the Lua programming language. Lua is a lightweight, imperative scripting language that is primarily used embedded in other programs and is notable for its extensibility and speed

The Battle for Wesnoth in Science The Battle for Wesnoth has garnered a small amount of attention in scientific research across various scholarly domains. Since it is an open source application, the game can be freely adapted, modified or extended to meet the given requirements. Individual evaluation methods can then be easily and modularly integrated. In addition, the game not only provides a fairly strong artificial intelligence that can play the game, but also allows it to be exchanged for any other artificial intelligence via the Wesnoth Markup Language. These factors, combined with the ability to execute custom code, provide a reasonable testbed framework.

For social studies, the game has been incorporated as an application for study cases on teaching methodologies [20], [27] or served as a platform for research on human-computer interaction in the modding community [45]. The research focus directly related to the game is in the area of computational intelligence and revolves around the design and development of artificial intelligence [64], [5]. Hybrid learning in particular has been the most successful method so far, combining evolutionary learning and reinforcement learning to create a two-tier approach to artificial intelligence decision making [65], [40], enabling agent teams to play the game effectively [39], or even evolving strategies for a team of agents [37].

3 Content Generation Rolling Horizon Evolutionary Algorithm

This chapter introduces the concept and implementation of our proposed algorithm, the Content Generation Rolling Horizon Evolutionary Algorithm (CGRHEA). The following sections first provide a general conceptual overview of the task at hand. Thereafter, the design of all essential parts of our algorithm and their integration into the turn-based strategy game The Battle for Wesnoth are discussed.

3.1 Concept and Overview

The theoretical concepts behind CGRHEA and their discussion in the context of the desirable properties for PCG are presented in this section. This allows the classification of the algorithm and the derivation of design constraints for the implementation based on the goals of this work.

3.1.1 Concept

Parts of the map need to be generated considering the current game state and presented to the player, therefore the map is generated incrementally. Gamestates can change significantly between turns and require new map sequences, which is the reason why the algorithm adjusts its evolutionary search for the best map accordingly. During this process, the necessary area of the game map is constantly evolved in a short period of time. In order to keep up with speed constraints and not disrupt gameplay for too long, the genome dimension of individuals needs to be limited, as this reduces the calculations required by the algorithm. Consequently, the area necessary for generation is limited to the number of tiles that a player's units can reach and see in a single turn. The considered approach constitutes an online Search-based Procedural Content Generation algorithm.

3.1.2 Desirable properties

We discuss the fundamental properties mentioned in Section 2.1 for a PCG algorithm in the context of our design decisions in order to categorize our approach in the broader context generation characteristics.

Speed Normally, the player is used to being able to control their units as soon as the other side has finished their turn. But with our online map generation, we need to generate and update parts of the map between turns to avoid a player exploring an area that hasn't been generated yet without making the player wait too long. Considering that Battle for Wesnoth is a turn-based and not a real-time strategy game, players should generally be fine with waiting a few seconds, while there is also a map scrolling animation once a side ends the turn, but all longer than this should not be acceptable.

Generation must be completed in five seconds or less to avoid disrupting gameplay for too long. In addition, the time required must scale accordingly with the horizon size, since fewer tiles to be generated result in fewer necessary computations overall.

Reliability Our algorithm is search-based approach and uses the stochastic metaheuristic of EAs, therefore the generated result is not deterministic. The algorithm also takes into account the current state of the game being played, which means that the result also depends on states that the algorithm cannot influence. In terms of not producing content that negatively affects gameplay, the selected tile set does not contain any tiles that would prevent the player from playing the game.

The design ensures that only valid tiles and villages are placed by the generator and therefore all generated maps are always playable.

Controllability As the Evolutionary Algorithm is strongly characterized by its hyperparameters, this enables some control over the search process. However no reliable or deterministic control can be achieved due to the stochastic nature of our process mentioned in the reliability section.

Because this work focuses on fairness and speed, rather than generating as many different scenarios as possible, controllability is dropped to some extent to ensure fairness. The hyperparameters of the EA should influence the generated results and make a difference, which represents a form of controllability.

Expressivity and Diversity Due to the fact that the algorithm only considers a small part of the possible tileset that the game offers, diversity and expressiveness of generated results is definitely limited. But the fitness function attempts to address this issue by discouraging small changes in local parts of the map to avoid pure randomness, and encouraging large-scale changes to keep the map interesting. As a consequence of the random initialization and the non-deterministic nature of the search process, no map will look the same as a previously generated one.

Full diversity is not achievable since not all possible maps can be generated as non-playable maps are excluded by design. In theory, the Evolutionary Algorithm would be able to do this, but constraints and fitness severely limit this. Due to the stochastic nature of the search process, anything can be generated and expressed within the limits of the constraints. Running the algorithm on the same setup, i.e. the same hyperparameters, leads to different, and therefore divers, results each time. In terms of the necessary limits from the constraints and fitness, the algorithm is expressive and divers.

Creativity and Believability In terms of emulating human-designed content, our generator produces very different content compared to a human, since the focus of our work is on speed, reliability and flexibility based on the game state. Humans usually design maps with an intention, because they have the expected gameplay in mind. The Evolutionary Algorithm only reacts to the gameplay, but does not predict it. In comparison, humans create a much wider range of maps by having more constraints that are less restrictive than the EA.

The algorithmic creation of game maps of Evolutionary Algorithms represents a creative process in a broader sense, as unexpected unpredictable solutions emerge, an essential aspect of creativity. The algorithm incorporates fitness aspects by design to generate a map that, when examined visually, displays a believable map.

3.1.3 Overview

The algorithm script is configured via Wesnoth Markup Language to call a preloading routine to initialize all necessary computational objects, which is the first initialization of the Evolutionary Algorithm and Rolling Horizon. This happens before the round starts and the user sees anything on the screen. In addition, the main script is executed waiting for the start of each turn of any side running the EA with the Rolling Horizon in the *start turn* function. As each side is treated separately, the main script first determines the current playing side and whether the termination criterion that the entire map is generated is met. If this is not the case and the current rolling horizon is not empty for the upcoming side's turn, the evolutionary search process runs for a set amount of generations, *#generations*. This includes the typical evolutionary processes of individual selection for reproduction, *select population*, the application of recombination operators, *crossover parents* and *mutate child*, fitness evaluation, *compute fitness*, and final selection for the next generation via *select environment*. Thereafter, the best individual is selected and the map tiles in the current horizon are updated based on the individual's height map encoding in *update map*. To make the map a bit more visually appealing, the function *generate overlay* randomly creates forests in a certain height range and a short road network. Map generation is then complete for the current side's horizon, but to prepare for the next turn change, we update the global information of the farthest unit on the other side whose next turn it is in *update horizon*, to predetermine the rolling horizon there. Moreover we update the number of already generated tiles necessary for the Layout Entropy Fitness. The current side's player or AI then gains control and can play their turn, and once the turn is ended, the previously described process is repeated for the other side. Algorithm 2 displays these information in a short and compressed form.

Algorithm 2 Content Generation Rolling Horizon Evolutionary Algorithm

```
procedure cgrhea;
begin
    while game runs and generation not finished do
        start turn
        if not empty(current horizon) then
            for #generations do
                select population
                crossover parents
                mutate child
                compute fitness
                select environment
            end for
            update map
            generate overlay
        end if
        update horizon
    end while
end
```

3.2 Content Representation

We represent the in-game map in The Battle for Wesnoth as a heightmap in our algorithm because "Both textures and many aspects of terrains can fruitfully be represented as two-dimensional matrices of real numbers" [66]. The heightmap is a discrete grid in which each cell stores its surface elevation data, a value that is then mapped to a graphical representation in game. The Battle for Wesnoth defines maps in its own map file format, which includes metadata in the file's header and raw map data as multiple lines of terrain code strings. A line represents a row in the hex grid, and terrain code strings are shorthands for the actual graphics that are loaded by the game engine and displayed in the end. They contain two characters for the base terrain layer and can contain two additional characters for the terrain overlay. The Battle for Wesnoth provides easy-to-use map access and modification functions via a Lua API, where map data can be accessed using simple two-dimensional coordinates.

Inspired by the standard map generator in The Battle for Wesnoth, which uses only a small palette of tiles, the height values are mapped to four different possible terrain codes: Medium Shallow Water, Green Grass, Regular Hills, and Regular Mountains. Additional overlays used in the application are "Vh" for Cottage, which represents the default village, and "Fp" for Pine Forest. Table 3.1 provides an overview of all possible height value ranges and the resulting computational representation in the algorithm. The height ranges are not split evenly to give the generated terrain a more natural look. Since hills and mountain ranges are not intended to dominate the entire map, their range and therefore frequency is reduced in favor of more grassland across the map.

height range	terrain name	code string
0.0 - 0.3	Medium Shallow Water	"Ww"
0.3 - 0.65	Green Grass	"Gg"
0.65 - 0.8	Regular Hills	"Hh"
0.8 - 1.0	Regular Mountains	"Mm"

Table 3.1: Height value ranges and corresponding terrain representations.

For tile-based generation, the obvious approach is to use patches of multiple tiles as they have a generally higher expressiveness and are easier to combine to create interesting formations as less variation is possible. We justify our decision of using 1 by 1 patches using the example of a map with 40 by 40 tiles, 1600 tiles in total, the same size as maps created by the built-in map generator of The Battle for Wesnoth. Figure 3.1 shows a comparison of tile patches of different sizes. A 3 by 3 patch covering 9 tiles would take up half a percent of the map, resulting in less than 200 patches for the entire map compared to 1600 tiles each for 1 by 1 patches. So increasing the patch size by two in both dimensions, the total tiles to be generated will be reduced by a factor of 8 in our example. Additionally, since the generated map will dynamically expand even to latitude 1 or 2 if units only move that much, larger patches could not be used for it, or at least require edge case coverage for it. Furthermore, the Differential Tile Fitness already accounts for larger neighborhoods for steepness and the mutation operator additionally changes not only individual tiles but also tiles around them. Although the smallest unit is a 1-to-1 patch, in many parts of the algorithm larger parts are considered in terms of generation. However, maps with small patches are difficult to generate, especially with larger creative structures, and are less resilient to turbulence, requiring careful design of evolutionary operators. In any case, our approach could include larger patches to address the previously mentioned weaknesses of smaller patches, while also covering edge cases for smaller maps. Nevertheless, our basic approach for the proposed algorithm as a proof of concept only deals with the smallest possible unit tile of 1 by 1.

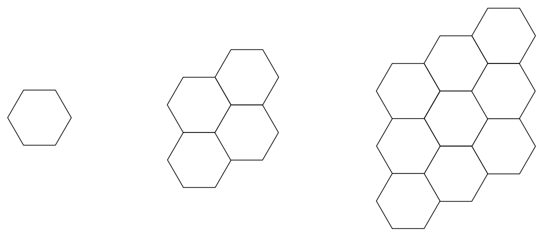


Figure 3.1: Comparison of hexagonal tile patches of size 1x1, 2x2 and 3x3

3.3 Fitness Functions

The following sections introduce the considered fitness functions for our proposed algorithm, justify their application in the context of map generation, and discuss the implementation aspects in detail. All fitness functions listed should be minimized for optimization.

3.3.1 Differential Tile Fitness

Inspiration Observable patterns in nature, for example leaf patterns or other appearing to be random patterns, have a kind of smoothness property. Points that are close together usually look very similar to each other, while points that are far away can look highly different. This constitutes a relation linking positional proximity to similarity where changes in the local environment are gradual while a global scale they can be large. Applying this reasoning to height maps, given their spatial proximity, a given tile should have the same height as its neighbors Metaphorically, this can be compared to the steepness of mountains. If you climb a mountain and walk a short distance at one point, the path does not appear very steep, but the entire track still has incline and the valley is at a lower elevation than the summit. In differential calculus, the gradient, a generalized derivative of a multivariate functions, can be interpreted as local changes that describes the degree of slope or steepness. Inspired by this analogy, we designed a fitness function to measure the steepness of our heightmap, since it is a discrete multidimensional function where its gradient can be approximated with finite differences. We consider the steepness property in our context to be the magnitude of the gradient, meaning the magnitude of the partial derivative vector. The Differential Tile Fitness should sensitize the algorithm to smoothness at the local levels, therefore prioritize a map with varying tile patches which are locally smooth.

Implementation As a measure of the steepness in our generated map, we compute the Differential Tile Fitness over finite differences as an approximation for the height gradient of each tile in the heightmap. For a hexagonal tile map, each tile has six immediate neighbors, and for each of these six directions we include the partial derivatives, plus second-level neighbors to cover larger features and their smoothness in the map as the second order derivative. For the second-level neighbors, six are located in the same direction as the first-level neighbors and their derivative can be computed via differential quotients, but this leaves six more between those to be treated separately. To account for these, we simply take the average of the adjacent derivatives.

By definition, the heightmap is a discrete function, i.e. a function where the domain and range are each a discrete set of values, rather than an interval in $\mathbb{R}$ as for continuous functions. The difference quotient, inspiration for the Differential Tile Fitness, is usually applied to continuous functions to define their derivative. But the difference quotient is not limited to continuous functions, which is why we apply this general principle to the two-dimensional discrete function defined on a hex grid, the heightmap.

The standard definition of the difference quotient ϕ for a function $f(x)$ and two points x_0 and x_1 is as follows:

$$\phi(x_1, x_0) = \frac{f(x_1) - f(x_0)}{x_1 - x_0} \tag{3.1}$$

For continuous functions, the derivative for the function at a given point x_0 is calculated by letting the difference $h = x_1 - x_0$ approach 0 by calculating the limit $\lim_{h \to 0} \phi$, this is called the differential quotient. Since we limit our approach to height maps, in the following part we define a differential quotient D_f for two-dimensional hex grids and will refer to this definition below when mentioning the differential quotient. Since the smallest possible limit in our discrete function is one step towards the nearest neighboring tile, we define the point x_0 as the tile whose Differential Tile Fitness is determined and x_1 as a point in its first-level neighborhood N_1. This leads to six points that meet this criterion, and therefore six differential quotients, two in each of the three hexagonal dimensions. To combine these into one value, we take inspiration from the Euclidean norm and consider the six differential quotients as six variables of the gradient vector. Because a norm is a mapping that associates a mathematical object with a number that is intended in some way to describe the size of the object, this is sufficient for our purposes. In other words, the norm of the differential quotient represents the magnitude of the partial derivative vector in the hexagonal map. As the distance between x_0 and its neighbors is indeed one, we always consider $x_1 - x_0$ to be one and can omit the denominator in Equation 3.1, resulting in the following equation for the differential quotient:

$$tD_f(x_1, x_0) = f(x_1) - f(x_0) \tag{3.2}$$

where $f(x)$ represents the associated height of tile x in the heightmap. Then the first order derivative f_{DT1} is the norm over all differential quotients of x_0 in its first-level neighborhood N_1, formulated as:

$$f_{DT1} = \sqrt{\sum_{i=1}^{|N_1|} D_f(x_i, x_0)^2} \tag{3.3}$$

If a tile in the neighborhood of x_0 is not part of the map, it will not be included in the calculation, effectively setting its value to zero.

The basic differential quotient is designed for N_1, but in a similar way to how the approximation of the first-order derivative is calculated over a one step forward finite difference, the approach can be extended to higher-order derivatives. In this work, we consider the second-order derivative and therefore include tiles in the second-level neighborhood N_2 to calculate the steepness of a given tile x_0. We consider the second-order forward finite difference to approximate the second-order derivative as follows:

$$\phi_2(h) = \frac{f(x+2h) - 2f(x+h) + f(x)}{h^2} \tag{3.4}$$

where h is the step size between the previously given points x_0 and x_1, calculated as $h = x_1 - x_0$ and again limited to the smallest possible value. Again the denominator vanishes because $h^2 = 1^2 = 1$, resulting the second-order differential quotient:

$$\begin{aligned} D_{f2}(x) &= f(x+2h) - 2f(x+h) + f(x) \\ D_{f2}(x_2, x_1, x_0) &= f(x_2) - 2f(x_1) + f(x_0) \end{aligned} \tag{3.5}$$

Since all hex tiles are equidistant from each other, the principle of $h = x_1 - x_0$ leading to $x_1 = x_0 + h$ can be applied to the second-level tile x_2 as $h = x_2 - x_1$, resulting in $x_2 = x_1 + h = x_0 + 2h$. Similar to the first order, the direct second order norm is calculated as follows:

$$\hat{f}_{DT2} = \sqrt{\sum_{i=1}^{|N_1|} D_{f^2}(x_{ij}, x_i, x_0)^2} \tag{3.6}$$

where $x_{ij} \in N_{2j}$ is the tile in the uneven second-level neighborhood, which is located in the direction from x_0 to x_i. This includes the six second level neighbors mentioned above that are in the same direction as those of the first level.

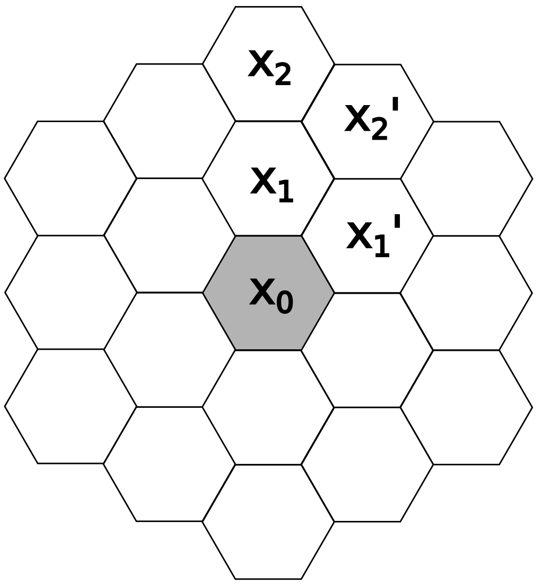


Figure 3.2: Involved tiles in the one second-level differential quotient.

For the six remaining tiles, each lying in the even second-level neighborhood N_{2k} between two covered tiles, an ambiguous computation exists in the second-order forward finite difference, since two tiles, x_1 and x_1', can both be taken on the way from x_0 to x_2. In the following we refer to these intermediate tiles as $x_{ik} \in N_{2k}$, where each tile x_{ij} is associated with the tile $x_i k$ clockwise next to it. To account for this ambiguity, we calculate the average between the norm for both possible finite differences $D_{f^2}(x_{ij}, x_i, x_0)$ and $D_{f^2}(x_{ik}, x_i, x_0)$. The second order derivative f_{DT2} is calculated by including the second-order forward finite difference for the remaining tiles $x_{ik} \in N_{2k}$ into the calculation of equation 3.6, as shown in Equation 3.7, where x_i and x_i' are the two tiles lying on the path between x_0 and x_{ik}. Figure 3.2 displays all involved tiles for the calculation of one second-level differential quotient.

$$f_{DT2} = \sqrt{\sum_{i=1}^{|N_1|} D_{f^2}(x_{ij}, x_i, x_0)^2 + \frac{D_{f^2}(x_{ik}, x_i, x_0)^2 + D_{f^2}(x_{ik}, x_i', x_0)^2}{2}} \quad (3.7)$$

For each tile, we define its steepness as the sum of the norm of the first and second derivative in its neighborhood on the height map $f_{DT} = f_{DT1} + f_{DT2}$ and express this as a Differential Tile Fitness value. As the biggest possible height difference between two tiles is one, the possible values for f_{DT1} range from zero to $\sqrt{6 \cdot 1^2} \approx 2.45$. Likewise, the largest possible value for f_{DT2} is $\sqrt{12 \cdot 2^2} \approx 6.92$ if x_0 and all tiles in N_2 have height zero and all tiles in N_1 have height one, or vice versa. To constrain our fitness scores, we divide the results by their maximum, $\sqrt{6} + \sqrt{48} \approx 9.37$, to preserve values in the range of zero to one. To account for all generated tiles in the horizon, the Differential Tile Fitness of an individual is the sum of the Differential Tile Fitness values for all tiles divided by the number of generated tiles.

3.3.2 Layout Entropy Fitness

Inspiration As already mentioned that two distant points can look very different, this becomes a necessary property for map generation to satisfy variety in the generated content. If every point on the map is smooth, meaning rendered as the same tile, there is no point of interest to search for in the map, and no need to continuously generate at all if everything looks the same from start to finish. But by enforcing diversity on a larger scale, we can guide the generation to create new structures that are not yet present on the map. In a figurative sense, the smooth map described represents a high-order map, since all points on the map are of the same type, because they are equally distributed. Mathematically, this can be expressed via entropy, a general measure of order and distribution based on the observation model presented, here the position of tiles in the map. To provide a counterforce against the smoothing factors of the Differential Tile Fitness, we designed a Layout Entropy function that needs to be minimized, since low positional entropy is achieved when we have low order in the map, which means that the tiles of the same type are distributed differently across the map. In the overall scheme, Layout Entropy Fitness should guide the search algorithm to find solutions that have several different areas of interest at larger scales.

Implementation The concept of entropy in information theory was introduced by Shannon [50] in 1948. The entropy H of a discrete random variable X with n possible values is defined as

$$H(X) = -\sum_{i=1}^{n} p_i \log_2(p_i) \tag{3.8}$$

where p_i is the probability of the value $x_i \in X$. The maximum value of entropy is $\log_2 n$ when all probabilities p_i are equal. In general, entropy is an order metric, where high entropy corresponds to low order and vice versa, but order and disorder are strongly dependent on the observation model. The observation model defines the attributes to be observed and their quantization and thus influences the weighting and appraisal of the observed parameters.

To determine the degree of order in the generated map, we need to measure whether all tiles are of the same type and equally distributed, which would represent high order and low entropy. This case is not desirable for our map, and since low fitness represents high quality for our EA, we need to invert the computed entropy value used as the fitness score. Another necessary adjustment is the scaling of possible entropy values, since we have four possible tiles that can be generated, the maximum possible entropy would be $\log_2 4 = 2$. To keep these entropy values in the zero to one range, we simply divide the calculated value by the maximum possible entropy. Equation 3.9 coherently represents these adjustments, where our Layout Entropy Fitness f_E is the inverse of the entropy for all tiles, represented as their count $t_i \in \{Water, Grassland, Hills, Mountains\}$ in the current horizon, scaled in the range from zero to one.

$$f_{LE} = 1 - \frac{-\sum_{i=1}^{4} t_i \log_2(t_i)}{log_2 4} \tag{3.9}$$

3.3.3 Village Fitness

Inspiration Since villages are an important part of the game mechanics, driving the circle of demand between units and resources that keeps the game interesting and flowing, the normal approach of offline generators is to place them reasonably evenly on the map. We chose to generate them each turn along with the normal base terrain layers as this allows for a base form of dynamic game adjustment. Due to the fact that an expanding map is linked to the expansion of one side's units, an abuse approach could be to move units as far as possible in order to generate more map for their own side compared to the enemy. Therefore, this side could dominate a larger part of the map since it can capture the villages generated there and still have villages left near its base that are also safe to capture. This would be the dominant strategy compared to first securing the villages near the player's keep. To keep multiple exploration strategies viable for the game, our algorithm considers the expansion of each side and places villages based on their ratio. In general, a player with a larger explored area should have captured most of their villages, or at least be confident of conquering them in the future, and therefore have more resources on hand compared to a smaller explored area. This should result in fewer villages being generated in the next turn for the player with the larger exploration area and more for the other side. To try to keep the game fair, the number of villages placed for each side needs to be monitored, especially in the early stages. If one side gets significantly more villages, they are heavily favored to win the match due to the self-reinforcing circle of demand. Therefore, our established Village Fitness function favors individuals that represent a balanced village distribution over all sides. The Village Fitness function represent the main driving force for a basic form of dynamic difficulty adjustment, which is primarily focused on resource equity over the course of the game.

Implementation As aforementioned, the overall goal is to keep the game balanced, which requires a balanced distribution of resources in the form of villages on the map for all sides. The size of the generated tiles for each side as a measure of their expansion is included in the fitness in addition to the number of villages. Overall, Village Fitness assigns individuals low scores that represent good quality when their proposed number of villages represents a reasonable distribution in the context of the current game state. If one side has explored a larger part of the map compared to the other, fewer villages should be generated since we can assume that in previous horizons a sufficient number of villages were already generated. For the other side of this scenario, Village Fitness should favor individuals with comparatively more villages. We express this as the absolute value of the distance between the village horizon ratio, which represents the percentage of villages in the current horizon, and the side area ratio, the inverse ratio of the generated areas.

First, the village horizon ratio vhr is calculated as the quotient of the scaled number of villages $k \cdot nv$ in the current horizon and the size of the current horizon c_h. If the calculated fraction is greater than one, we limit the value to a maximum of one. Similar to the map generator for The Battle for Wesnoth, which generates a village roughly every 40 tiles or 2.5% of the map, the village horizon ratio scales the number of villages by a factor of $k = 10$ to avert placing too many villages. Therefore, the maximum value is already reached when 10% of the horizon is covered with villages, which already represents an above-average village density.

$$vhr = min\left(\frac{n_v \cdot k}{|c_h|}, 1.0\right) \tag{3.10}$$

Second, the side area ratio sar is calculated as the number of generated tiles for the opposite side divided by the sum of all generated tiles, meaning the current ga_{cs} and opposite side. This inversion, where the area of the opposite side is included in the dividend rather than that of the current side, accounts for the fact that overall fitness should favor solutions with fewer villages, represented by a lower village horizon ratio, when the opposite side has less generated tiles than the current playing side.

$$vhr = \frac{ga_{os}}{ga_{cs} + ga_{os}} = 1 - \frac{ga_{cs}}{ga_{cs} + ga_{os}} \tag{3.11}$$

The resulting Village Fitness is calculated as:

$$f_v = |vhr - sar| \tag{3.12}$$

By taking the absolute value of the difference between vhr and sar, we achieve the set design goal of balancing the number of villages for each side based on each side's explored area in the game. Regardless of whether too many or too few villages are generated, we achieve overall higher fitness, which represents a lower overall quality for the evaluated individual. Since the ratio of village horizon and side area is in the range from zero to one, the resulting Village Fitness does not have to be further scaled.

3.3.4 Weighted Sum

As mentioned in Sections 2.1, 2.2.1 and 2.2.3, a single-objective fitness function is typically insufficient, particularly in the context of Procedural Content Generation, where multiple criteria affect the perceived quality of the generated content. Following this line of reasoning, we decided to integrate several fitness functions into a single fitness value via weighted sum combination. The apparent and most popular multi-objective Evolutionary Algorithm is NSGA-II [6], a fast non-dominated sorting approach that excels in efficiently solving constrained multi-objective problems. Since the Lua interpreter in The Battle for Wesnoth does not allow the inclusion of external code modules, we be required to reprogram the entire algorithm. Additionally, this approach is computationally expensive and therefore time intensive, which is not best suited to our fast and reactive approach to map generation. For these reasons, the integration of NSGA-II was discarded in favor of a faster and simpler weighted sum approach.

3.4 Rolling Horizon Evolutionary Algorithm

Battles on favorable terrain and careful planning of unit paths are extremely important for the game, as multiple areas of terrain open up different paths to strategically important objectives. But terrain tiles vary greatly in their strategic value, as mountains and deep water, for example, are only beneficial to a few unit types, while grassland tiles have the lowest movement costs, meaning they can be traversed quickly. Therefore, the map defines the available resources and their reachability by placing the villages directly on the map and generating the tiles around them. Due to the circle of demand, precautions are required for the placement of available resources for each side, especially if the map is created as the game is played. Fairness should be a factor to consider, otherwise one side will outclass the other due to the difference in available resources, but one benefit of creating the map mid-game is the ability to adjust earlier misplacements later in the game.

3.4.1 Evolutionary Algorithm

As this algorithm tries to find the best possible map in a short amount of time, the population consists of a small number of individuals, represented genetically as a heightmap and a list of villages. The height map is randomly initialized, meaning that each tile is assigned a randomly assigned value between zero and one. Each tile has a 0.6% chance of getting a village placed on it as an overlay. Their fitness is assigned based on the aforementioned weighted sum of three fitness functions, differential, entropy and village based. Individuals are selected via tournament selection with tournaments of size two to reduce the dominance problem and encourage consideration of different types of maps without making a full random selection.

For the crossover of two individuals we developed a special form of one-point crossover that takes two previously selected individuals and produces two children. Since we have expanding map circle pieces, the rolling horizon, we split these at a random vertical point and switch between parents. This is to keep the operator simple in a one point crossover manner, but still allow for clipping out larger parts that are locally adjacent so as not to lose all effort on local smoothing compared to randomly selecting the same number of tiles at all possible locations in the horizon. Villages are copied from parents to their children as they are mainly modified in the mutation operator.

Another problem-specific operator was developed for the mutation. When a tile is randomly chosen to mutate, all other neighbors within a radius of two are mutated. Basically, each adjacent tile is adjusted to the mutating tile, or rather the difference in height between the mutating tile and the neighbor for the inner circle with a factor of 50% and for the outer neighbors with 33%. Mutation of just a single tile seemed counterintuitive for such large tile structures, where the rolling horizon can consist of more than a hundred tiles, as it would be an imperceptible change with overall mutation chances in the single-digit percentage range. Increasing the mutation probability parameter would result in only small local changes that are randomly distributed across the map and could only tear apart the height map landscape as they do not depend on the previous mutation. Therefore, in order to introduce a smoothing factor into the map that takes local properties into account, we decided to reverse the intuition that a tile is influenced by its neighbors and instead its neighbors are affected by the selected tile. As this affects multiple tiles rather than just one and is kept local, the mutation probability can also remain low, which satisfies the commonly understood criteria for mutations and brings advances in the generation process as smooth maps are favored in the Differential Tile Fitness function. This operator does not necessarily have to be smoothing all the time, considering that a single steep tile is picked in an otherwise smooth neighborhood, the mutation will levitate the smooth neighborhood to its own level, reducing overall smoothness but potentially increasing entropy, as mountain ranges can grow from grasslands, a scenario that keeps the map interesting and is encouraged by the Layout Entropy Fitness.

In the same way a base layer tile can mutate, so can a village overlay but in this case either a village is randomly added to a random tile, or an existing village is deleted, or a randomly existing village changes its position. Since villages are the main resource on the map, we consider their numerical balance to be more important than their specific location. When a village is generated in the next horizon, the horizon is usually not extremely wide, since the width is based on the movement and view range of each side's farthest unit, so a generated village is always in acquiring range for the associated side, regardless of position. But the total generated is of paramount importance as they heavily influence how the game further progresses. An imbalance should be avoided when both sides are on even ground, therefore the mutation operator only adds or removes one village instead of several, since generating multiple villages with one mutation for one side represents an instant economic boost for that side. This has to be compensated for on the other side in future horizons, which could lead to a general swing back and forth in compensation measures, which is not desirable. For that reason, only incremental changes are made, while Fillage Fitness tends to favor the most appropriate village spread for the current game state.

3.4.2 Rolling Horizon

Inspired by the Rolling Horizon Evolutionary Algorithm principle presented in Section 2.2.2, we derived our own rollout principle applicable to tile-based map generation in the context of the game The Battle for Wesnoth. In the following part, we briefly present the main differences between the original RHEA and our proposed version.

Because turn-based strategy games do not require real-time input, players are used to a slower game pace where they wait for their opponent to finish their turn after taking actions themselves. Also, since the game pans to the other side in an animation when a turn changes, there is no need to complete the generation in milliseconds when playing, but a strong recommendation not to take more than a few seconds so as not to keep the player waiting too long.

Now that our use case considers sequences of the map to be generated rather than action sequences, we need to define our horizon and how much we want to roll out with it. Since larger areas generate more computational effort, we have opted for the minimum area with the lowest computational effort for calculation and evaluation. The central reference point is the starting keep for each side, since each side has its own separate rolling horizon that is independent of the other. From that point, the farthest unit determines how many tiles to generate in each direction, using the keep as the center of the growing circle. To this distance d_{fu} of the farthest unit we add its movement and sight range. This includes any tiles that the player might see when moving their furthest unit as far as possible in one turn, since more distant points are obscured by the fog of war. We call this annulus the rolling horizon for each side and calculate it before the start of each turn. By generating a growing circle of tiles in each direction, we simplify the generation process to not miss small tile spaces that are never visited by units during a turn. As players explore farther from their keep for enemies to defeat and villages to capture, this usually results in both sides meeting at some point in the middle of the map when the keeps are on opposite sides placed on a square map. At this point, both horizons have each covered almost half of the map, which almost completes the map generation most times, ensuring that the game can then be played as if a previously created map had been loaded. An advantage of map generation compared to action sequence evolution is that tiles do not have to be generated each time. Players usually have to take action each turn, but if all units do not move further than the farthest unit has already gone, the horizon does not expand as everything they might see is already covered by the generated map, so no further computation needs to be done. When the map generation is finished, usually before the round is even remotely finished, no further search process needs to be started, therefore our algorithm only delays the gameplay in the first few turns for adequate map sizes.

Originally designed to evolve action sequences with a strong focus on real-time decision-making by limiting computation time to just a fraction of a second, we derived a suitable version for map generation in turn-based games. This version differs greatly from the original version in fundamental aspects. For example, time is the most obvious factor, as we computing much longer, but also the difference between action sequences and tile sequences. While we do not have a game state as a sequence of actions that is updated iteratively, we do have a map state for the game. For this map state, the algorithm continuously decides which tile is placed in the current map sequence. Therefore, sequences of the map are transformed into each other by the respective action of defining their tiles. There is no forward modeling involved as previous horizons cannot be used to forward propagate a new map state for the next sequence. If the map were forward modeled, all individuals would keep a larger sequence of tiles and iteratively evolve them. Then, each round, the best individual would decide on the next tiles for the necessary area of the player and optimize the remaining tiles for the upcoming turns. This represents are desirable process that is closer to the originally designed Rolling Horizon Evolutionary Algorithm. But for all other individuals in the population, the forward-modeled map is likely to deviate greatly from the best individual. In the original version, they would just keep developing their own sequence, but actions are different than tiles. Since smoothness and fairness rely on a map made up of coherent parts, the previously decided map sequence would render all other individuals as mismatches. Therefore, only the best individual and their offspring would remain competitive in terms of fitness, which is tantamount to evolving only a single individual through the entire search process. Evolving just a single individual and effectively discarding the rest of the population each time seems counterproductive and counterintuitive. For these reasons, we decided to discard the forward model and have each individual genetically represented as only the necessary map sequence for the next turn. Nonetheless, the fitness functions are incremental and incorporate some form of back-coupling from the previous generation, since prior decisions remain on the map and are also evaluated. Village fitness accounts for previously placed villages and Differential Tile Fitness covers earlier placed tiles on the overlapping edges.

Although the original version and our derived version may differ greatly in certain theoretical aspects, the original version would not allow for efficient map generation in our case. But since the general process was a great inspiration and we kept our approach close to this principle, we decided to keep the name Rolling Horizon Evolutionary Algorithm.

4 Evaluation

In this chapter, the basic parameter settings, various methods and metrics are presented to evaluate the proposed Content Generation Rolling Horizon Evolutionary Algorithm. A set of experiments is conducted to determine whether the stated hypotheses are supported by the results in order to answer the aforementioned research questions of this work.

4.1 Hypotheses

The goal of this chapter is to analyze the generation process statistically, to determine and to evaluate the best set of hyperparameters for the algorithm. The evaluation focuses on the following hypotheses:

- The algorithm has high sensitivity to hyperparameters in a sense that small parameter changes lead to strongly varying results.
- The map will be generated each turn in no longer than 5 seconds.
- The algorithm generates fair maps.

4.2 Experimental Setup

All experiments and tests are carried out on a computer running with an AMD Ryzen 9 3950X 16-Core Processor (2,2 GHz), a NVIDIA GeForce GTX 970 and 64GiB DDR4 RAM (2,4 GHz). The software runs on Ubuntu 20.04.4 LTS (Focal Fossa) with The Battle for Wesnoth Version 1.16.2.

4.2.1 Data Acquisition

The Battle for Wesnoth Lua interpreter does not provide access to file manipulation due to security reasons but this is a vital feature for experiment data acquisition. To get around this problem we made use of the logging and error functionality that The Battle for Wesnoth offers. Instead of writing to dedicated specially created files, the experiment results are thrown as error because errors are logged in log files on Windows or in the terminal for the Linux application where they can be written from to the disk.

4.2.2 Game Setup

The experiments are conducted in a standard The Battle for Wesnoth scenario with two sides playing one against one. Each side is controlled by the standard artificial intelligence provided by the game itself. Similar to the included "Random map" generator from the game, the map size is fixed at 40 by 40 tiles with set starting locations for each side in the top left and bottom right corners of the map. An experiment run corresponds to a game round until the entire map with 1600 tiles is generated. As the advancement of the generation mechanism is based on units and their exploration, the same experiment can take different numbers of turns to finish, since the artificial intelligence never plays a round that is similar to the other. Because of this, experiments stop at different points and do not have a fixed number of turns.

4.2.3 Metrics and Hyperparameters

Hyperparameter Evaluation The hyperparameter sensitivity of the algorithms is to be investigated, which is why we determined the most influential hyperparameters of the EA, namely the mutation chance, the population and mating size and the weighted sum values. We chose these parameters because they are likely to have a strong impact not only on the generated result, but also on the computation time and general incentive of the search process. Starting from a hand-picked selection of values for these parameters, we modified them sensibly to obtain a set of rational parameters to examine their influence, summarized in Table 4.1. This results in four parameters with three values each, so in total 3^4 possible hyperparameter combinations with 31 runs per parameter combination, i.e. a total of 2511 runs. Mating size values are given as fractions, but the nominal resulting value for the algorithm is always rounded to the nearest rational power of two. Values of the weighted sum address the three fitness functions in order of Layout Entropy, Differential Tile, and Village Fitness. The *#generations* for which the Evolutionary Algorithm

hyperparameter	value 1	value 2	value 3
mutation chance	0.01	0.05	0.005
population size (μ)	20	10	30
mating size (λ)	$1/8$	$1/4$	$1/2$
weighted sum set	($1/3$, $1/3$, $1/3$)	(0.15,0.7,0.15)	(0.7,0.15,0.15)

Table 4.1: Hyperparameters and their respective assigned values.

runs each turn is fixed at 12. We define the fitness of one experiment run as the average fitness over all turns of the game round where a horizon was rolled out and tiles were generated. This excludes turns where no further generation was necessary either because the whole map is generated or the furthest unit and its movement plus vision range were not greater than the previous one. The fitness of a single turn is represented by the fitness of the best individual in the current population.

Mann–Whitney U Test The Mann-Whitney U test [34] is a nonparametric test that is commonly used to test for difference in sampled random variables. It was originally designed to test whether, for two populations X and Y, there is an equal probability that a randomly selected value x from one population X is greater or less than a value y chosen at random from the other population Y. However, the test and its result are rarely interpreted in this exact way. Most of the time, we interpret a significant p-value as meaning of differences between the distributions or whether two independent samples originate from the same distributions. In other words, the Mann-Whitney U test tests for differences between two groups on a single, continuous, ordinal variable with no specific distribution. Therefore, the formulation would now be that the distribution X underlying sample set x is the same as the distribution Y underlying sample set y. Consequently, under the null hypothesis H_0 the distributions of both populations are identical and the alternative hypothesis H_1 is that the two distributions are not equal. As with any other significance test, we assume that H_0 holds and that both distribution sample sets are equal if p $> \alpha$.

In practice, the Mann-Whitney U test is applied as a nonparametric alternative to the unpaired t-test [56] when its assumptions are not met. This is true when the data to be tested is ordinally scaled, or generally when there is no prior knowledge of the distribution of the data. This applies to the distributions of fitness values from different hyperparameter sets for our proposed algorithm, since EAs are strongly nonlinear due to the fitness function alone.

Horizon-time Ratio The above reasons for variations in turn lengths for the experiments also cause horizon sizes to vary greatly per turn and per round. In order to have meaningful discussions about the speed constraints, the horizon-time ratio is briefly introduced here. We define the horizon-time ratio as the quotient of the computing time for this turn and the number of generated tiles per turn, the horizon size. This ratio is used as a measure to describe how long the entire evolutionary search process spent searching and deciding on each generated tile in the horizon.

4.3 Experiments

The following list of experiments is conducted to examine the performance of CGRHEA and to test the hypotheses formulated in Section 4.1. A brief overview of their intentions and approach is presented, while the results thereof are shown and discussed in the following section.

Hyperparameter Sets To determine the optimal hyperparameter set, the single best values for each hyperparameter are determined. By fixing a value for one hyperparameter and varying all possible combinations for all others, we get a distribution of fitness values defined by that hyperparameter. The comparison of the median overall fitness of this distribution with the other possible values of the hyperparameter provides information about their influence on the optimization process and their ranking among each other. The experiment concludes with the determination of a best set of parameters, which is further evaluated and explored in the subsequent experiments.

Hyperparameter Performance The median fitness performance determined in the previous experiment is not meaningful enough to unequivocally determine the optimal hyperparameter set. It is important to examine whether the fitness distribution of an optimal set is also an independent distribution compared to other sets. The set is optimal if changing one or more hyperparameter values also leads to different fitness distributions. Then these specific hyperparameters have a significant impact on the quality of the generation. The Mann-Whitney U test is applied in this experiment to verify this. The p-value for determining whether the null hypothesis H_0 is proven set to $\alpha = 0.05$. Thus, if two sets of hyperparameters achieve a p-value greater than or equal to 0.05, their distributions are identical and therefore have no significant influence.

In a form of tournament, every possible combination of two sets of hyperparameters is tested with the Man-Whitney U-test and the resulting p-value is assigned to them.This enables an overall ranking of all hyperparameter sets based on the number of null hypotheses rejected versus another set and the sum total of all assigned p-values throughout the tournament.

Turn-based Performance The algorithm runs over several turn and has to output a sequence of the map almost every turn. Each output result is valuable to gameplay as it affects the immediately following turns and persists for the rest of the game. Due to these reasons, the assessment of the best individual's fitness progression over all turns is important. The experiment determines whether the algorithm is able to consistently output high-quality results or whether it fluctuates by monitoring the fitness of the best individual over all turns for each turn. Additionally, the average fitness across the game is provided as a measure of the expected performance by the algorithm. In addition, the maximum and minimum fitness values achieved in each round are displayed together with the average score to give an overview of the possible fitness range of solutions.

Generation-based Performance This experiment extends the introspection of the evolutionary search process and examines the fitness progress of each generation in a single turn. For each experiment run of the best hyperparameter set and each turn in that run, the fitness of the best individual in each generation is measured. This metric provides insight into the search progress and whether the algorithm is able to produce higher quality solutions with more time or is stuck in a local optima. The fitness values and their average are displayed analogously to the previous display, but over the course of all generations.

Speed Analysis To verify the hypothesis that the map is generated in no more than 5 seconds each turn, the experiment analyzes the speed of the algorithm. First, the distribution of the horizon-time ratio is evaluated to measure how long it takes to generate a single tile and whether that time differs across the course of the game. Then the elapsed time for all experiments runs over all turns compared to the constraint is displayed along with the average time taken.

Fairness The village fitness function is the main part of the algorithm that promotes fairness by ranking individuals based on the ratio of villages placed to area generated for each side. The experiment checks whether the generated villages of each round balance out for both sides playing the game. This is visualized for a randomly selected example experiment run. In addition, the total number of villages is added to give an overview of the development of the village count over several turn for all experiment runs.

4.4 Results

In this section, the results and observations of the various experiments are evaluated. The following sections cover each experiment and its results in detail before summarizing all results in a final discussion to verify the hypotheses formulated at the beginning of this chapter.

4.4.1 Hyperparameter Sets

The initial experiment compares the fitness score distribution for a given hyperparameter with different values. As shown in Figure 4.1 for the mutation chance parameter, the foremost parameter value is 0.01 with a median of 0.128. The other two values show poorer fitness performance and a wider interquartile range, which represents the spread of the data. Notably, while 0.01 is the best parameter, it also has the most outliers. A lower mutation chance for the individuals leads to less change in the map compared to the initial random initialization of the height map. Therefore, map sequences appear random rather than smooth, which is penalized by Differential Tile Fitness. The entropy layout fitness that encourages changes in the map may only slightly offset this disadvantage. On the other hand, a much higher mutation chance leads to the

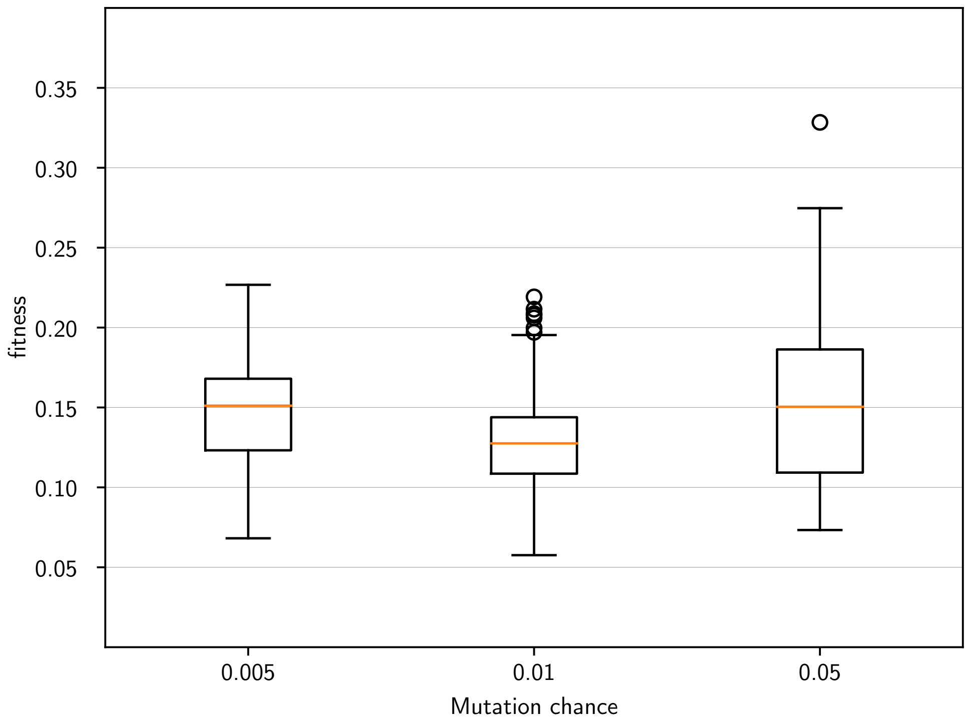


Figure 4.1: Distribution of fitness values for the assigned mutation chance values.

opposite scenario of a truly smooth map with low Differential Tile and high Layout Entropy Fitness. The value 0.01 represents a middle ground between these two scenarios, trying to balance smoothness and entropy. This can lead to very different results, but if no middle ground can be found, neither Differential Tile nor Layout Entropy Fitness are satisfied at all. The large number of outliers is a strong indicator for this assumption. In particular, the longer upper and lower whiskers and larger quartiles overall indicate strongly differing results in the map.

Figure 4.2 shows the results for the population size parameter. The distributions are very similar overall, almost the same in all terms. An increase in population size appears to downshift fitness scores only slightly, but a significant difference is difficult to verify. All medians are around 0.14 with similar standard deviation, number of outliers, and spread of the data. The popula-

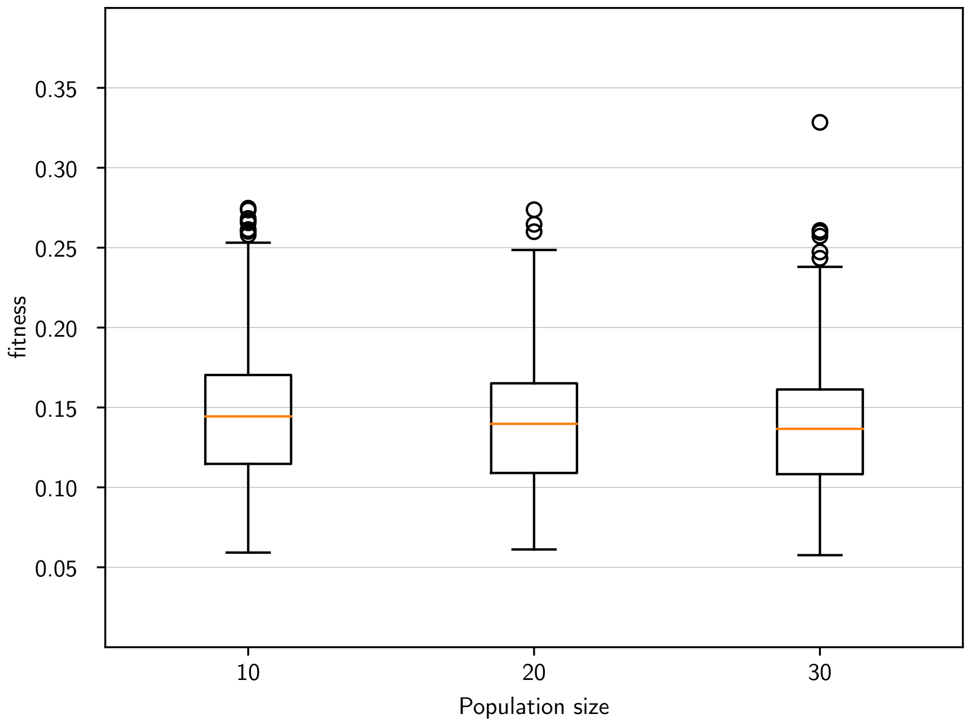


Figure 4.2: Distribution of fitness values for the assigned population size values.

tion size results can be considered almost predictable due to the limited search framework with a fixed number of generations that CGRHEA represents. With Evolutionary Algorithms, a larger population size usually results in there being more average individuals that do not contribute to the search process. On the contrary, they even prevent the algorithm from searching faster for better results. Each indivdual needs to be evaluated by the fitness functions, which typically takes the most computation time. More individuals may also reduce the selective pressure for high-quality individuals. In this time-sensitive application that needs to converge quickly and find high-quality solutions fast, every superfluous individual can represent an impediment. Nevertheless, the increase in population size can lead to better solutions, as the second experiment shows only marginally.

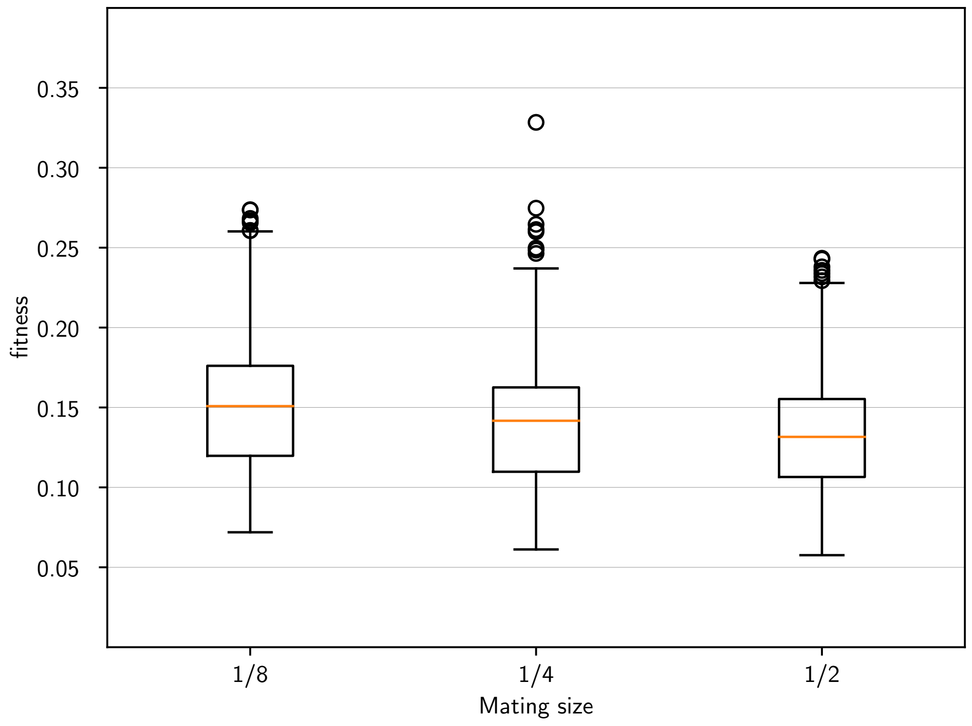


Figure 4.3: Distribution of fitness values for the assigned mating size values.

Analogous results are available for the mating size parameter, as can be seen in Figure 4.3. Increasing mating size decreases the median fitness scores achieved by the best individual. The impact of change on this parameter can be seen more clearly, although the general distribution of the data does not change. Larger mating sizes increase selection pressure and the overall chance of retaining more high-quality individuals for the next generation. In addition, this leads to a broader search through the search space and ensures that outliers are reduced as the children produced can replace them in the population. Overall, the parameter appears robust with similar quartiles, whiskers, and outliers, but increasing mating size makes a noticeable difference compared to population size.

The results for the last hyperparameter are displayed in Figure 4.4. The weighting sum shows one of the most disparate results because it not only greatly affects the resulting fitness, but drives the overall search in a certain

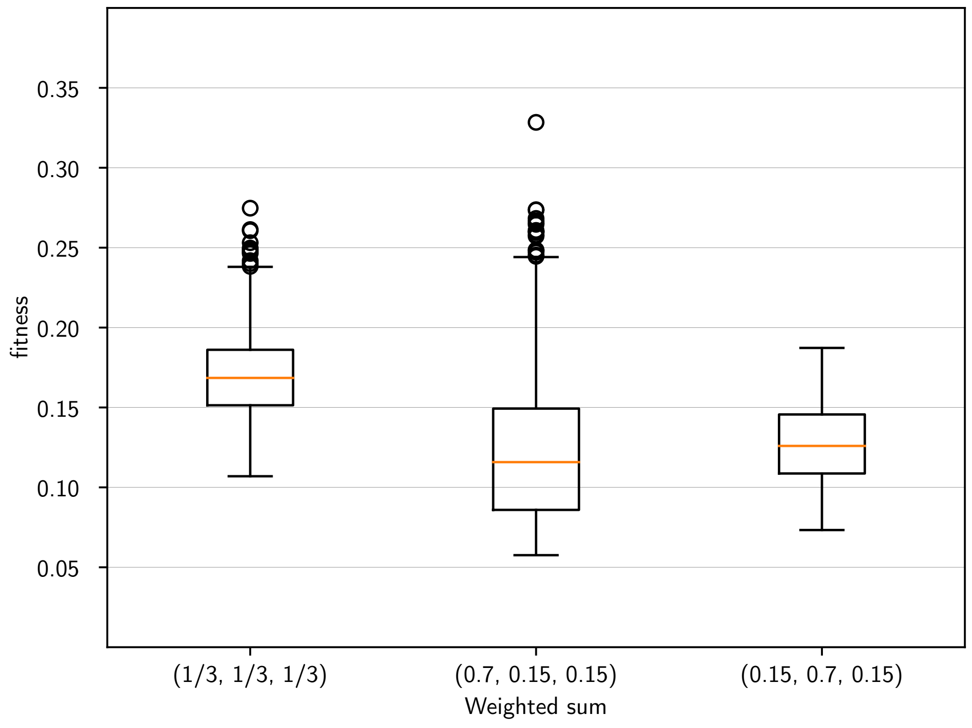


Figure 4.4: Distribution of median fitness values for the assigned weighted sum value sets.

direction based on one fitness function being weighted differently than the other. The jack-of-all-trades approach of weighting each fitness function 1/3 performs the worst in terms of median fitness, producing many outlier solutions. In general, it is already difficult to satisfy three different criteria, but optimizing all of them with the same importance seems almost impossible intuitively. This is also visually represented in the results, either it works and a robust solution is found in the interquartile range, or the solution is likely an outlier and performs much worse. In comparison, the second value set, which gives more weight to entropy and less weight to the other two, achieves lower median fitness. However, this takes a toll on the robustness, since quartiles are visibly larger and outlier solutions are just as worse as those of the first set of values. Plotted with the highest standard deviation of 0.046, focusing on entropy shows that a change in the map can lead to better or worse outcomes than the first case, but does not express robust control over it. Only marginally worse in median fitness, but way upfront in terms of robustness, the final approach focuses on the different tile fitness. No outliers are generated and the interquartile range is similarly close to the first set of values while also having the shortest whiskers. Individuals may not achieve the same minimum fitness scores as the second value set, but they remain competitively close to them in all cases. This combination of low median fitness and standard deviation of 0.023 pushes it above the second value set in the ranking as the best weighted sum parameter.

	median ± standard deviation		
hyperparameter	value 1	value 2	value 3
mutation chance	0.151 ± 0.036	0.128 ± 0.03	0.15 ± 0.046
population size (μ)	0.144 ± 0.041	0.14 ± 0.039	0.137 ± 0.038
mating size (λ)	0.151 ± 0.041	0.142 ± 0.038	0.132 ± 0.036
weighted sum set	0.168 ± 0.027	0.116 ± 0.046	0.126 ± 0.023

Table 4.2: Median and standard deviations for the fitness values of each hyperparameter.

Table 4.2 displays the results for all hyperparameters again in a compact overview. The median already shows an optimal value for two hyperparameters, the mutation chance and mating size. Additionally, the standard deviations for these two are quite similar, so we choose 0.01 for the mutation probability and $^1/_2$ for the mating size as the most optimal parameters for our algorithm. The remaining parameters are fairly close in terms of median fitness. As previously mentioned, the weighted sum is determined by the significantly better combination of median fitness and standard deviation. For the population size, the standard deviations are also almost the same. The decision is still quite ambiguous, but here we take the numerical best of 30 as the optimal parameter. Therefore, the best hyperparameter set is set as [0.01, 30, $^1/_2$, (0.7, 0.15, 015)].

4.4.2 Hyperparameter Performance

The results of the hyperparameter performance experiment are shown in Figure 4.5 as a heatmap. Accordingly, the results of the tournament held are presented in Table 4.3. For the main diagonal, all values are 0.5 because the Mann-Whitney U test identifies a set of parameters as coming from the same distribution compared to itself. In addition, the heatmap is symmetric since the results of testing a set of hyperparameters X vsersus Y are the same as testing Y versus X. The heat map shows only a few clusters with values distinctly above the set value of $\alpha = 0.05$, which accepts the null hypothesis H_0. Therefore, we can conclude that most sets appear to differ from each other in terms of their distribution of fitness scores.

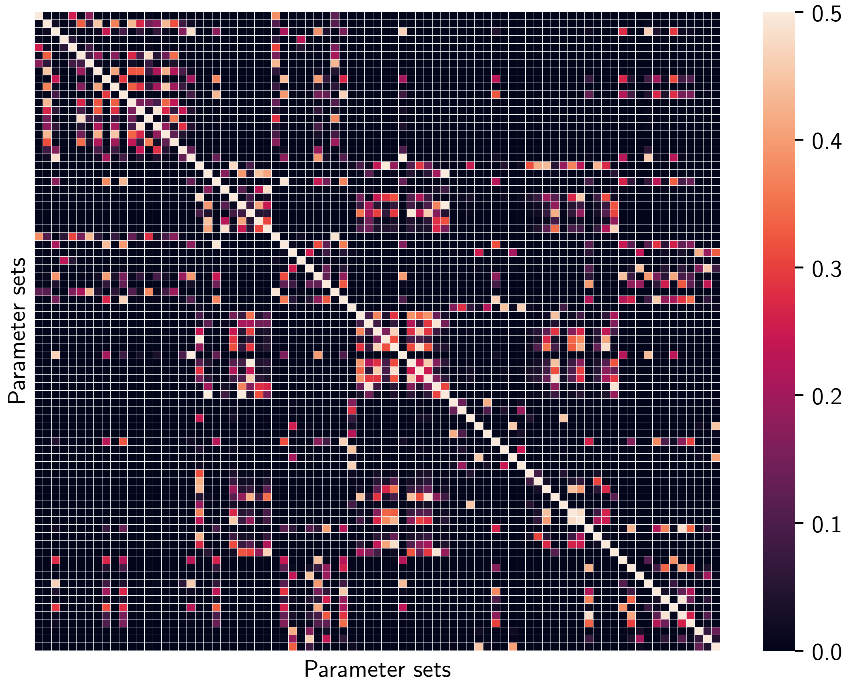


Figure 4.5: P-values of all parameter set combinations for the Mann-Whitney U test.

The resulting tournament winners show that our previously determined best parameter set is represented in the top five but in second place, notably behind first place. The first place set represents a combination of the most outlying parameters in terms of fitness performance. Since the Mann-Whitney U test only considers distributions, we can conclude that this set produces greatly differing results and fitness scores, probably the overall worst. Another point to note is that with 81 parameter sets in an all-vs-all tournament, the maximum number of wins possible would be 80 since a set can never win against itself. Parameters such as population and mating size have already been shown to perform similarly with almost the same median fitness scores, regardless of their assigned value. Therefore, it is not surprising that the tournament winners still lost to sets with only different population and mating sizes, as they have no overall impact on the fitness scores achieved. The Mann-Whitney U test accounts for this by rendering them as stemming from the same distribution, resulting in a defeat.

parameter set	count	p-value sum
$0.05, 10, 1/8, (0.7,\ 0.15,\ 0.15)$	78	0.325
$0.01, 30, 1/2, (0.15,\ 0.7,\ 0.15)$	78	0.777
$0.01, 10, 1/2, (0.15,\ 0.7,\ 0.15)$	78	0.834
$0.05, 10, 1/8, (1/3,\ 1/3,\ 1/3)$	77	0.367
$0.01, 20, 1/8, (0.15,\ 0.7,\ 0.15)$	77	0.788

Table 4.3: Mann-Whitney U test tournament results.

Overall, one could determine the optimal set of hyperparameters based solely on tournament results by counting the times a given hyperparameter is present in the top five sets. Then the result would be almost the same since only the population size would be different at 10, but again the population size decision was previously ambiguous in terms of its lack of impact on fitness.

4.4.3 Turn-based Performance

The development of the average fitness of the best individual over the in-game turns, observable in Figure 4.6, shows that the fitness is kept at almost the same level. Remarkable differences are worth highlighting only at the beginning and end of the game. In the first turn of the game, a vast area has to be generated around each side's starting keep, which represents a tougher challenge for the algorithm and the reason that the fitness starts of higher. But as the game progresses, the necessary horizons to be generated become smaller and the fitness shows an adjustment of the algorithm towards more ordinary values. Higher fitness peaks start to frequently set in in the later turns. As the map sequence generation nears completion, only small chunks remain at the corners of the map. Although small in size, they are surrounded by a large amount of generated tiles that may not allow proper tile placement. For these cases it is to be expected that the Layout Entropy and Differential Tile Fitness in

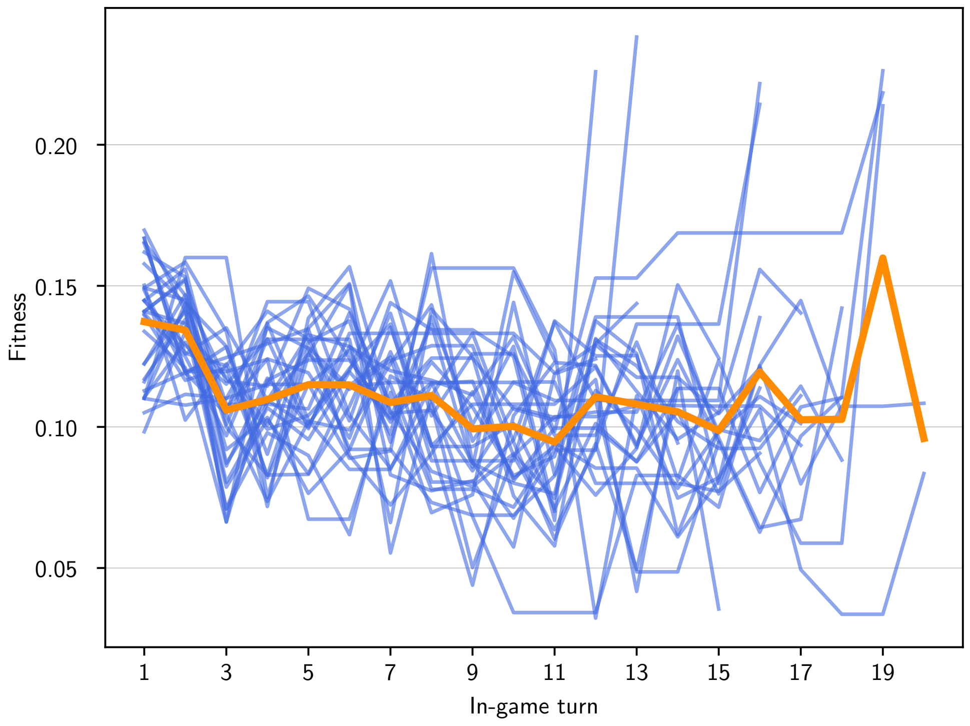


Figure 4.6: Fitness values and average score over all turns.

particular are higher. In terms of fitness drift, we can see that it is kept within a certain range for most of the game, while again spikes happen towards the end, which seems acceptable considering it only affects the corners of the map.

The comparison of minimum, average, and maximum fitness over the same turns, depicted in Figure 4.7, shows a similar trend for all three metrics. There is overall nearly constant fitness with a downward trend for the minimum and an upward trend towards the end for the maximum. Initially there are higher deviations due to larger horizons to be generated in the first turns. In the first half of the game, the graphs appear to be similar, only shifted to higher and lower values from the average. Significant differences in the graphs only appear towards the end, as the maximum fitness in particular seems to be heavily influenced by the ending scenarios for the last remaining tiles. Minimum fitness, on the other hand, seems to be able to keep up longer while keeping the fitness scores low. This indicates that there are ways to meaningfully solve the gener-

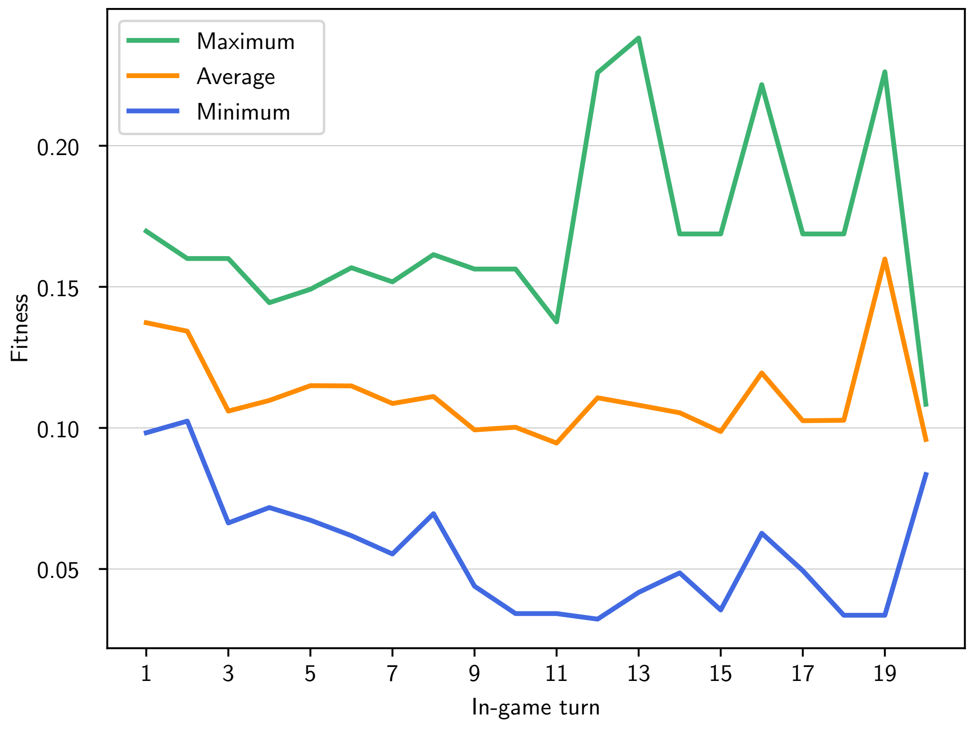


Figure 4.7: Minimum, average and maximum fitness values over all turns.

ation for the final tiles, indicated by the minimum, but these may be hard to find and it is generally more likely to generate worse solutions, indicated by the maximum fitness. In summary, the graphic comparison confirms the previous results, since a similar behavior can be observed. The plot starts with high initial fitness, a steady downward trend thereafter and strongly fluctuating fitness values at the end.

4.4.4 Generation-based Performance

The results of the fourth experiment, shown in Figure 4.8, represent an interesting trend when compared to the average fitness results across all turns in the previously shown experiments. In relation to the generations, a clear downward and thus optimization progress can be observed. Due to the random initialization of the heightmap, a very high fitness start for the individuals can be expected at the beginning. Thereafter, fitness declines at a nearly constant

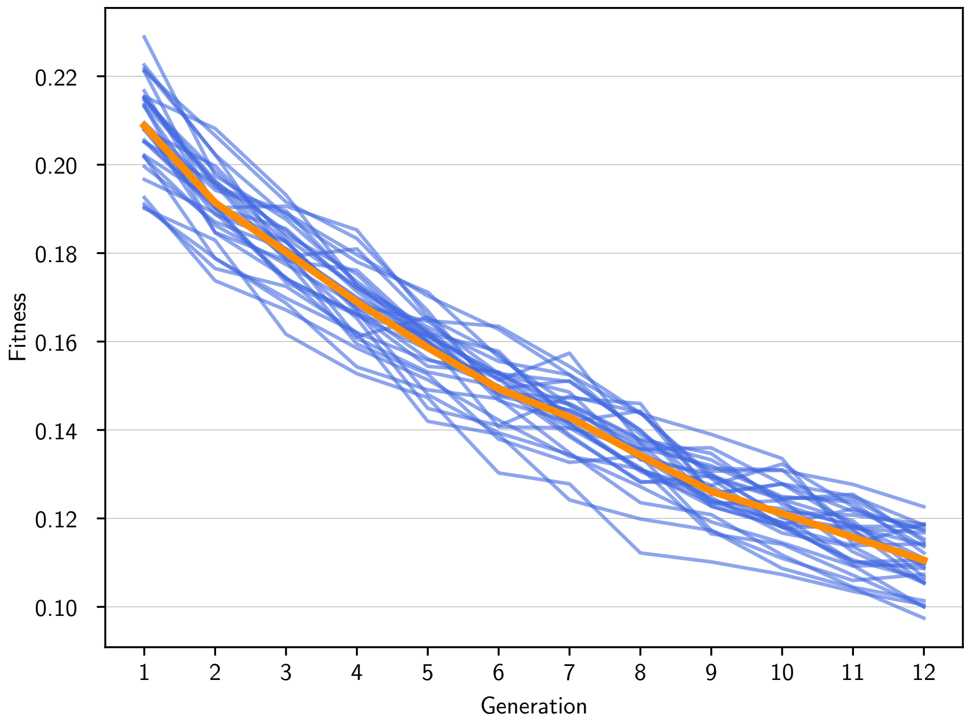


Figure 4.8: Fitness values and average score over all generations.

slope over each generation. It is important to note that these fitness scores are averaged over each turn, meaning they include the data for the very differing starting and final turns of the game. Nevertheless, the fluctuation range of fitness score remains almost the same across all turns and all generations. The observation that these fluctuations do not change much over all generations could indicate that the Rolling Horizon-inspired approach of our proposed algorithm is capable of improving any solution over the generations. The fact that the slope is nearly constant shows that the algorithm can tackle the optimization problem, but it clearly takes a certain number of generations to get suitable results. In addition, there is still the option to generate higher quality individuals when more resources or time are available.

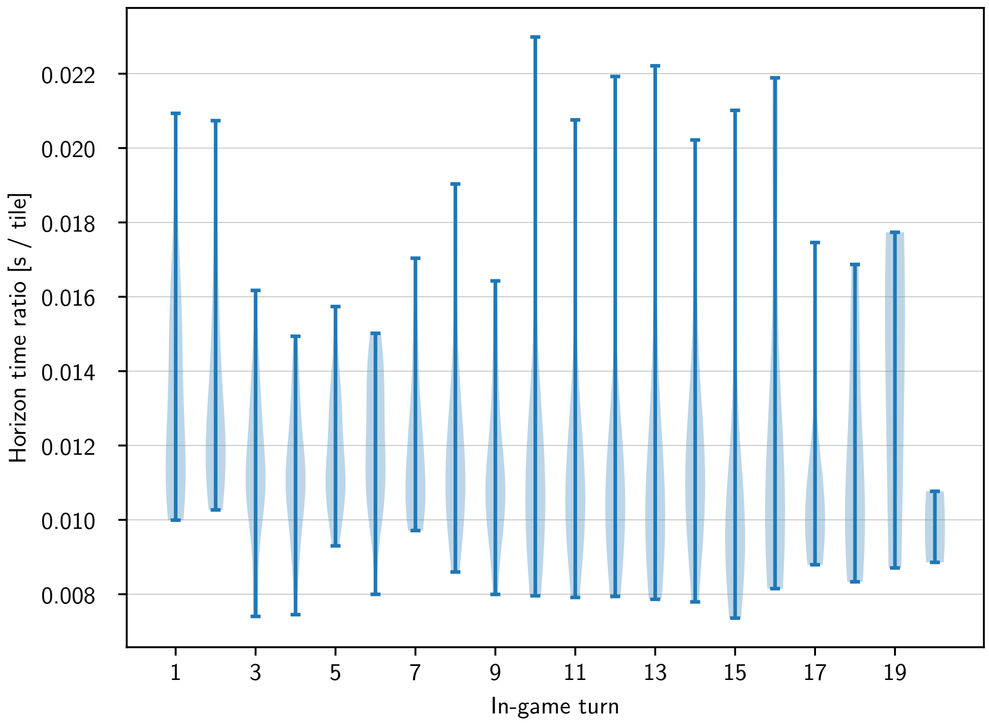


Figure 4.9: Distribution of horizon-time ratio values over all turns.

4.4.5 Speed Analysis

Figure 4.9 shows the distribution of the horizon-time ratio per turn and indicates that the time required to generate a tile is kept relatively equal throughout the game. Variations between turns only seem to appear on a lower scale, while all turns have some outliers towards a higher horizon-time ratio. Most of the data points are spread out at the bottom of the chart with a tiny downtrend across in-game turns differing by just a few milliseconds. Outliers can be subject to certain scenarios where the horizons can be larger and therefore more data needs to be processed or more memory allocated. Overall, the perception is that the average time per tile is stable with fluctuations within a short range. With an average horizon-time ratio of 0.012 seconds per tile over all turns, the algorithm takes 19.2 seconds to generate all 1600 tiles in the map. Since the game can take up to 20 turns, 10 turns per side, until the generation is complete, this results in about 1 second per turn for map generation. This calculation only applies if all the necessary tiles were distributed evenly across all turns, which is never the case in our scenario.

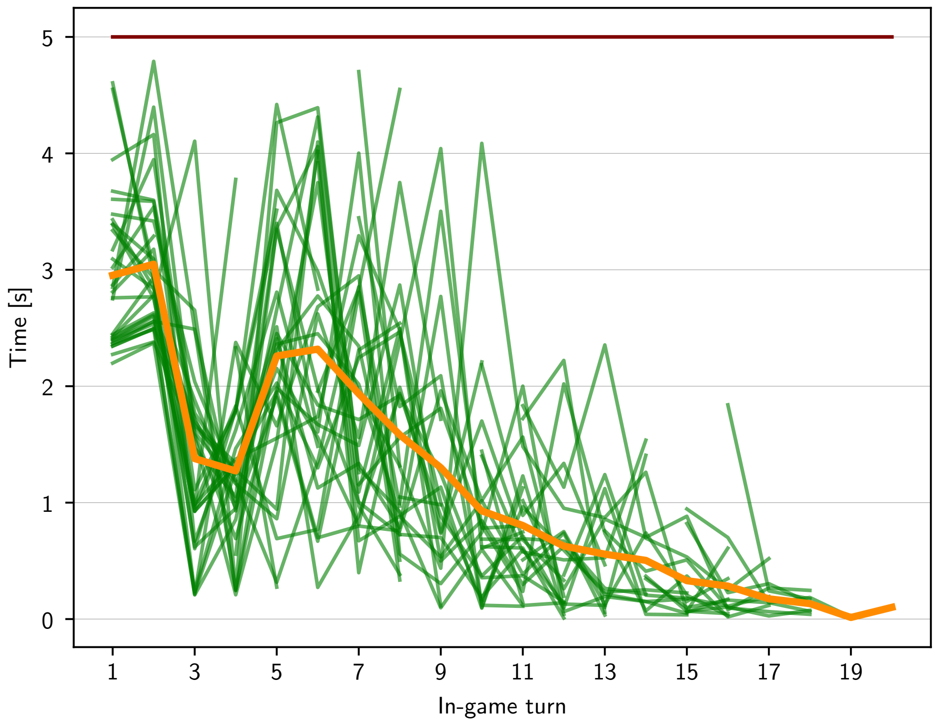


Figure 4.10: Time and average time frame over all turns.

The 19.2 seconds are therefore spread differently over the turns, as Figure 4.10 already shows. Due to the aforementioned larger horizon size at the beginning of the game, the generation runs considerably long in the first turns. Afterwards, the time per turn decreases as units recruitment starts but units cannot move in the turn they spawned on the map. Thereafter, each side usually tries to capture the first villages near their keep without exploring far into the map. The second spike reliably occurs around turns five through seven, when units start exploration in search of further resources to claim, leading to larger horizons and longer computing times. In the following turns, the time decreases almost linearly, since the initial portion generated at the beginning and the first exploration phase typically cover large parts of the map. The rest is only slowly explored as these tiles are furthest away from the keeps on either side, resulting in progressively lower average computing time per turn.

The rest is only slowly explored as those lie furthest away from the keeps of both side resulting in an ever going down average computation time per turn. Spikes in the graphs can occur when units high long movement ranges decide to explore as far as possible. As the horizon size for the next turn is determined by the movement and vision range of the farthest unis, this can create a lot of tiles to generate. The results clearly show that no run required more than five seconds per turn, with the overall average remaining under three seconds across all game rounds evaluated.

4.4.6 Fairness

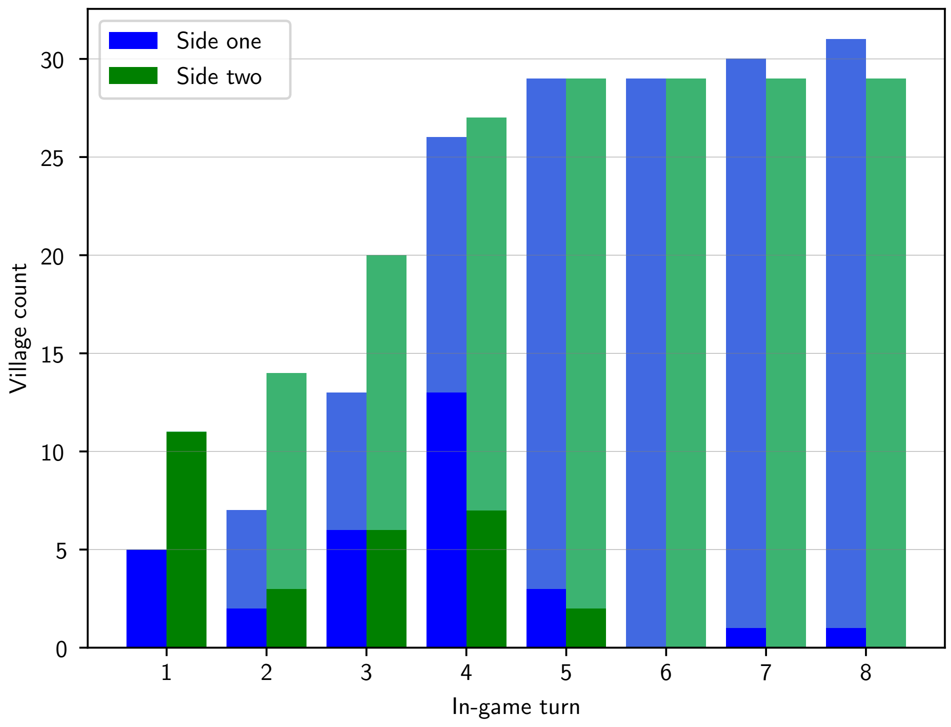


Figure 4.11: Village count and total number of villages for both sides over all turns.

The exemplary run of the fairness experiment in Figure 4.11 gives a telling overview of the general procedure of village placement on the map, which serves as the main factor for fairness. The horizon of side one is always the first one generated, where the algorithm places an ordinary number of villages. Thereafter, the second side starts the game at a disadvantage in generated tiles and villages, since no map was previously generated for this side. To compensate for this, the algorithm places more villages on the map sequence of the second side. In the following turns, the difference in the total sum of placed villages slowly balances out until they reach an almost equal state at the end. The number of villages generated changes significantly between turns and, the side that receives the most generated villages in each turns changes several times.

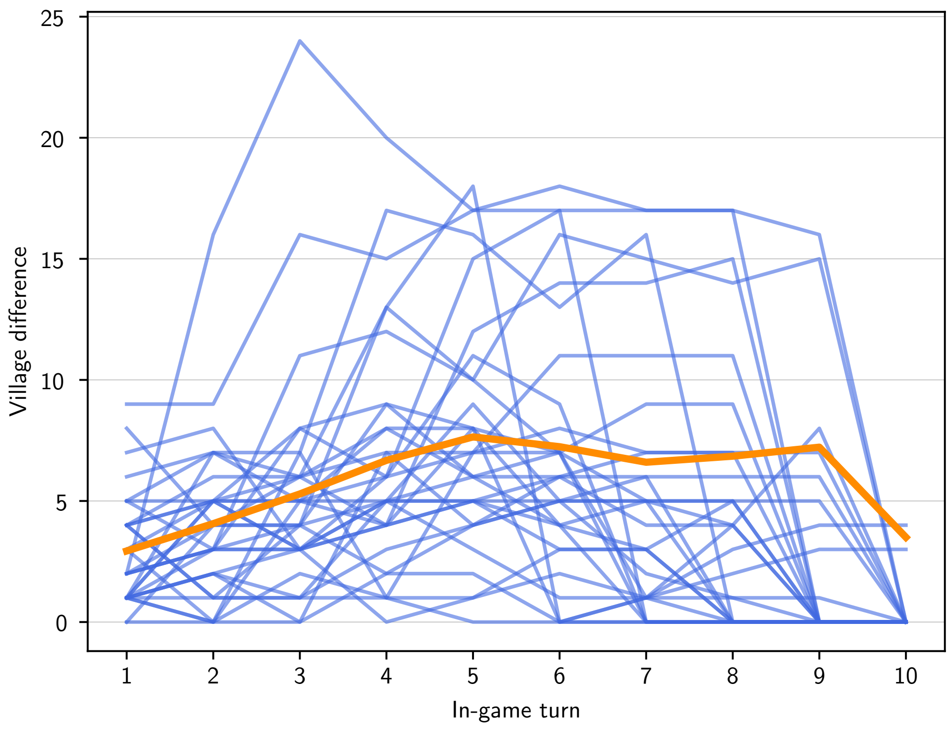


Figure 4.12: Village difference and average difference over all turns.

The village differences over all turns are kept small on average for both sides, but are not zero as shown in Figure 4.12. Therefore, perfect numerical balance is not achieved. Moreover, the first and last turns show fewer differences than the turns in between. In the first turns, the horizons are fairly evenly sized and there are not as many villages to place, making it less difficult for the algorithm. The algorithm then has several rounds to balance the total number of generated villages for both sides and therefore ends up with a smaller difference in the last round. The overall difference is between three and seven villages per game, with more runs below average and only a few outliers with high village differences pulling the average up.

4.5 Discussion

The series of conducted experiments provided thorough insights into the optimization process of CGRHEA from a variety of aspects. The algorithm was able to stay under the hard constraint of five second of generation time per turn. Not a single evaluated run violated this limitation, with the overall average staying under three seconds. Nonetheless, the definition of acceptable game break time for map generation remains debatable and highly application specific. For the considered game scenario of a turn-based strategy game, the times required for map generation are acceptable. However, coupling speed constraints to design decisions remains a difficult task because the design of recombination operators, and hyperparameter tuning in particular, do not allow for shorter or longer runs of the search algorithm. Not running long enough or applying only a very small mutational chance will produce random results, as the search process has been shown to require time and significant changes to the initial encoding of individuals. This trade-off between time and resources is typical for optimization problems, especially for online algorithms running in a time-critical environment.

Regarding the hyperparameter sensitivity, we could show with the experimental results that the algorithm is sensitive for the considered parameters, but not for all. Mutation chance and weighted sum parameters stand out as the most influential, given their different results in performance depending on their assigned values. Population size and mating size generally remain fairly ineffective, since generation occurs online in a short amount of time where greater or lesser genetic diversity in the population and offspring cannot have much of an impact. As the Man-Whitney U Test tournament showed, not all correlations can be decoupled between the hyperparameter sets and a small fraction of insensitivity remains. Nonetheless, the parameters are able to meaningfully guide the search process toward higher-quality solutions and varied results that are also able to satisfy the given constraints.

In competitive turn-based games, whoever makes the first move usually has an advantage, which is compensated for in some games, but not all. Seeing the oscillations in village placement, especially in the early turns of the game, could be interpreted that way. But this behavior was not intended when the Village Fitness function was designed. These fluctuations are probably the reason why the total village difference is always non-zero during the evaluated rounds. A difference in village count for one turn invokes a countermeasure in the next that is unlikely to completely erase the difference. Therefore, the algorithm applies a series of countermeasures that only stop once the map generation is complete. This shows that fairness in games can have many different aspects and that integrating intended fairness mechanisms into an optimization is a difficult task, but even more difficult to unequivocally assess fairness. True numerical balance in fitness would not even constitute fair play since not all rounds are equal. In certain scenarios it is not desirable to have village balance, but to favor the side that is at a disadvantage. The designed Village Fitness function tries to take this into account by considering the generated horizon sizes. The algorithm can keep the village count quite equal, but it remains unclear and difficult to verify whether these differences were justified based on the context of the game state or if they represent fluctuations that occurred during the search process. Overall, the results show that the algorithm is able to generate fair maps, albeit at a low level. In addition, the algorithm is able to stay under five seconds of generation time per turn and is highly sensitive to the hyperparameter values for mutation chance and weighted sum.

5 Conclusion

In this thesis, an online Search-based Procedural Content Generation algorithm for tile-based map generation was designed. The algorithm enables the continuous creation of map sequences during the game and reacts to changes in the game state. It represents a combination of the search-based generation approach combined with a novel rollout principle inspired by Rolling Horizon Evolutionary Algorithms. In order to incorporate all requirements for meaningful map generation in the optimization process, a combination of three fitness functions, each with their own specific intent, was applied. Differential Tile and Layout Entropy Fitness promote solutions that appear smooth locally but vary on a larger scale as an aspect of interesting but believable maps. The Village Fitness was designed in such a way that the generated map remains fair as well.

Several experiments were carried out to determine whether the algorithm can keep up with a given time limit, which hyperparameters influence the search process to which extent and whether fairness aspects are represented in the end. The results indicate that the algorithm generates maps in a reasonable time. An optimal set of hyperparameters was determined, which expressed the overall best performance in the evaluation. A low level of fairness was ascertained in the map generation process, which represents a dynamic adjustment to game state changes. The evaluation shows that focusing on the main aspects seems superior to balancing all possible generation characteristics. In a time-critical scenario, a trade-off has to be made between the resources allocated for evolutionary search and the computation time. Since PCG defines a list of multiple desirable properties, there is no limit to possible fitness features to be incorporated that can promote these aspects. But the combination and weighting of all these features constitutes the hardest challenge in this context. Even simply combining multiple fitness functions into a single one leads to its own set of problems, which again require a trade-off. In particular, that some functions tend to be optimized at the expense of others. Instead, one could apply a multi-objective approach that optimizes for multiple goals at the same time and finds the set of non-dominated individuals that have unique combinations of strengths. Nevertheless, the incorporation of these methods usually weighs on the speed constraint, and it remains questionable whether the benefits outweigh the costs in this case.

The Rolling Horizon-inspired approach to continuously generated sequences of the map as ever-expanding circles around the starting points of the game is the main reason online generation became possible in the first place. This line of thought could be taken further to only generate patches around the farthest unit, thereby reducing the considered tiles to be generated to the smallest possible amount. Further computations would be required to determine these areas at every turn, and improved edge case coverage by the fitness function would be needed, but these extensions appear beneficial. As the time necessary for the generation is not distributed evenly over the course of the game but is centered on specific turns, the specified number of generations for which the algorithm runs could be dynamically adjusted. This adjustment would allow finding higher quality solutions for turns that can spare more time because they have smaller horizons. For larger horizon, this could also be used to ensure time constraints are not violated, but represents a trade-off in fitness scores. The other hyperparameters could potentially be dynamically adjusted as well to react more sensitive to game state changes and promote different generation aspects in specific scenarios. The overall dynamic difficulty adjustment is an aspect addressed only at a low level by our proposed algorithm. Nonetheless, the general approach to the optimization problem and the game itself provide more than enough opportunities to further address this issue. A more sophisticated approach would include the specific positions of the captured villages or villages in the capture range of units. In addition, the units of each side and the gold flux can also be included, for example to model the current score in terms of fairness. The limited number of possible tiles that can be generated is also expandable as the game offers a long list of tiles to create more visually appealing or interesting structures for the player. Since tiles additionally have an impact on unit movement, this would address multiple problems at once, but requires more than just smoothness and entropy as a factor for believable maps. To conclude, the presented algorithm represents a novel search-based map generation approach that can be used to generate a playable, believable, and fair map for a turn-based game that dynamically adjusts to the gameplay.

Bibliography

[1] Thomas Bartz-Beielstein, Jürgen Branke, Jörn Mehnen, and Olaf Mersmann. Evolutionary algorithms. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 4(3):178–195, 2014.

[2] Robert Bridson, Jim Houriham, and Marcus Nordenstam. Curl-noise for procedural fluid flow. *ACM Transactions on Graphics (ToG)*, 26(3):46–49, 2007.

[3] Thomas Bäck and Hans-Paul Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1(1):1–23, 1993.

[4] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[5] Martín Dans Cervero. Ia battle for wesnoth. Master's thesis, Universitat Politècnica de Catalunya, 2016.

[6] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[7] Agoston E Eiben and Selmar K Smit. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31, 2011.

[8] Raluca D. Gaina, Sam Devlin, Diego Perez-Liebana, and Simon M. Lucas. Rolling Horizon Evolutionary Algorithms for General Video Game Playing. *arxiv:2003.12331*, 2020.

[9] Raluca D. Gaina, Simon M. Lucas, and Diego Perez-Liebana. VERTIGO: Visualisation of Rolling Horizon Evolutionary Algorithms in GVGAI. In *The 14th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 265–267, 2018.

[10] Raluca D. Gaina, Simon M. Lucas, and Diego Pérez-Liébana. Population seeding techniques for rolling horizon evolution in general video game playing. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 1956–1963, 2017.

[11] Raluca D. Gaina, Diego Perez-Liebana, Simon M. Lucas, Chiara F. Sironi, and Mark H.M. Winands. Self-adaptive rolling horizon evolutionary algorithms for general video game playing. In *2020 IEEE Conference on Games (CoG)*, pages 367–374, 2020.

[12] Alessio Gambi, Marc Mueller, and Gordon Fraser. *Automatically Testing Self-Driving Cars with Search-Based Procedural Content Generation*, page 318–328. Association for Computing Machinery, 2019.

[13] Martin Gardner. Mathematical games: The fantastic combinations of john conway's new solitaire game "life". *Scientific American*, 223(4):120–123, 1970.

[14] Ioseff Griffith. Generation, evaluation, and optimisation of procedural 2d tile-based maps in turn-based tactical video games, 2016.

[15] Erin J Hastings, Ratan K Guha, and Kenneth O Stanley. Evolving content in the galactic arms race video game. In *2009 IEEE Symposium on Computational Intelligence and Games*, pages 241–248. IEEE, 2009.

[16] I. Her. A Symmetrical Coordinate Frame on the Hexagonal Grid for Computer Graphics and Vision. *Journal of Mechanical Design*, 115(3):447–449, 09 1993.

[17] Vincent Hom and Joe Marks. Automatic design of balanced board games. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 3(1):25–30, 2007.

[18] Lawrence Johnson, Georgios N. Yannakakis, and Julian Togelius. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10. Association for Computing Machinery, 2010.

[19] Giorgos Karafotias, Mark Hoogendoorn, and Ágoston E Eiben. Parameter control in evolutionary algorithms: Trends and challenges. *IEEE Transactions on Evolutionary Computation*, 19(2):167–187, 2014.

[20] Kaido Kikkas and Mart Laanpere. Playful cleverness revisited: open-source game development as a method for teaching software engineering. In Jürgen Münch and Peter Liggesmeyer, editors, *Software Engineering 2009 - Workshopband*, pages 267–272. Gesellschaft für Informatik e.V., 2009.

[21] Theodore Kim, Nils Thürey, Doug James, and Markus Gross. Wavelet turbulence for fluid simulation. *ACM Transactions on Graphics (TOG)*, 27(3):1–6, 2008.

[22] Abdullah Konak, David W Coit, and Alice E Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability engineering & system safety*, 91(9):992–1007, 2006.

[23] Rudolf Kruse, Sanaz Mostaghim, Christian Borgelt, Christian Braune, and Matthias Steinbrecher. *Elements of Evolutionary Algorithms*, pages 255–285. Springer International Publishing, 2022.

[24] Rudolf Kruse, Sanaz Mostaghim, Christian Borgelt, Christian Braune, and Matthias Steinbrecher. *Introduction to Evolutionary Algorithms*, pages 225–254. Springer International Publishing, 2022.

[25] Ares Lagae, Sylvain Lefebvre, Robert L Cook, Tony Derose, George Drettakis, David S Ebert, John P Lewis, Ken Perlin, and Matthias Zwicker. State of the art in procedural noise functions. *Eurographics (State of the Art Reports)*, pages 1–19, 2010.

[26] Ares Lagae, Sylvain Lefebvre, George Drettakis, and Philip Dutré. Procedural noise using sparse gabor convolution. *ACM Transactions on Graphics (TOG)*, 28(3):1–10, 2009.

[27] Haiyuan Lee. The integration of information technology into high school english teaching in china. Master's thesis, University of Wisconsin-Platteville, 06 2018.

[28] John Levine, Clare Bates Congdon, Marc Ebner, Graham Kendall, Simon M. Lucas, Risto Miikkulainen, Tom Schaul, and Tommy Thompson. General video game playing. In *Artificial and Computational Intelligence in Games*, Dagstuhl Follow-Ups, pages 77–84. Dagstuhl Publishing, November 2013.

[29] Diego Perez Liebana, Spyridon Samothrakis, Simon M. M. Lucas, and Philipp Rohlfshagen. Rolling horizon evolution versus tree search for navigation in single-player real-time games. In *GECCO '13*, 2013.

[30] Aristid Lindenmayer. Mathematical models for cellular interactions in development ii. simple and branching filaments with two-sided inputs. *Journal of Theoretical Biology*, 18(3):300–315, 1968.

[31] Jialin Liu, Diego Pérez-Liébana, and Simon M. Lucas. Rolling horizon coevolutionary planning for two-player video games. In *2016 8th Computer Science and Electronic Engineering (CEEC)*, pages 174–179, 2016.

[32] Luczak and Rosenfeld. Distance on a hexagonal grid. *IEEE Transactions on Computers*, C-25(5):532–533, 1976.

[33] Tobias Mahlmann, Julian Togelius, and Georgios N Yannakakis. Spicing up map generation. In *European Conference on the Applications of Evolutionary Computation*, pages 224–233. Springer, 2012.

[34] H. B. Mann and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):50 – 60, 1947.

[35] Jean-Eudes Marvie, Julien Perret, and Kadi Bouatouch. Fl-system : A functional l-system for procedural geometric modeling. *The Visual Computer*, 21:329–339, 06 2005.

[36] David Maung. *Tile-based Method for Procedural Content Generation*. PhD thesis, The Ohio State University, 2016.

[37] Bentley Oakes. Practical and theoretical issues of evolving behaviour trees for a turn-based game. Master's thesis, McGill University, 2013.

[38] Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, page 301–308. Association for Computing Machinery, 2001.

[39] Sanjeev Paskaradevan. A hybrid agent architecture for learning good cooperative behaviours for game characters. Master's thesis, University of Calgary, 2012.

[40] Sanjeev Paskaradevan and Jörg Denzinger. A hybrid cooperative behavior learning method for a rule-based shout-ahead architecture. In

IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology, volume 2, pages 266–273, 2012.

[41] Amit Patel. Hexagonal grids. `https://www.redblobgames.com/grids/hexagons/`. Accessed: 2022-08-14.

[42] Ken Perlin. An image synthesizer. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, page 287–296. Association for Computing Machinery, 1985.

[43] Ken Perlin. Improving noise. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, page 681–682. Association for Computing Machinery, 2002.

[44] Eric Piette, Dennis J.N.J. Soemers, Matthew Stephenson, Chiara F. Sironi, Mark H.M. Winands, and Cameron Browne. Ludii - the ludemic general game system. In *Conférence Nationale en Intelligence Artificielle*, July 2019.

[45] Giacomo Poderi and David J Hakken. Modding a free and open source software video game:"play testing is hard work". *Transformative Works and Cultures*, 15(1), 2014.

[46] Mike Preuss, Antonios Liapis, and Julian Togelius. Searching for good and diverse game levels. *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8, 2014.

[47] Procedural content generation wiki. `http://pcg.wikidot.com/`. Accessed: 2022-07-09.

[48] Sebastian Risi, Joel Lehman, David B D'Ambrosio, Ryan Hall, and Kenneth O Stanley. Combining search-based procedural content generation and social gaming in the petalz video game. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[49] Konstantinos Sfikas and Antonios Liapis. Playing against the board: Rolling horizon evolutionary algorithms against pandemic. *IEEE Transactions on Games*, 2021.

[50] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, 1948.

[51] Adam M. Smith and Michael Mateas. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200, 2011.

[52] Gillian Smith. An analog history of procedural content generation. In *Proceedings of the 10th International Conference on the Foundations of Digital Games, FDG 2015, Pacific Grove, CA, USA, June 22-25, 2015*. Society for the Advancement of the Science of Digital Games, 2015.

[53] Gillian Smith, Jim Whitehead, and Michael Mateas. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):201–215, 2011.

[54] Sam Snodgrass and Santiago Ontañón. Experiments in map generation using markov chains. In *FDG*, 2014.

[55] James C Spall. *Introduction to stochastic search and optimization: estimation, simulation, and control*. John Wiley & Sons, 2005.

[56] Student. The probable error of a mean. *Biometrika*, 6(1):1–25, 1908.

[57] Hisashi Tamaki, Hajime Kita, and Shigenobu Kobayashi. Multi-objective optimization by genetic algorithms: A review. In *Proceedings of IEEE international conference on evolutionary computation*, pages 517–522. IEEE, 1996.

[58] The battle for wesnoth. `https://www.wesnoth.org/`. Accessed: 2022-07-22.

[59] Julian Togelius, Mike Preuss, Nicola Beume, Simon Wessing, Johan Hagelbäck, and Georgios 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.

[60] Julian Togelius and Noor Shaker. *The search-based approach*, pages 17–30. Springer International Publishing, 2016.

[61] Julian Togelius, Noor Shaker, and Mark J. Nelson. *Introduction*, pages 1–15. Springer International Publishing, 2016.

[62] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation. In *Applications of Evolutionary Computation*, pages 141–150. Springer Berlin Heidelberg, 2010.

[63] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011.

[64] Daniel Wehr and Jörg Denzinger. Mining game logs to create a playbook for unit ais. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 391–398, 2015.

[65] Stephen Wiens, Jörg Denzinger, and Sanjeev Paskaradevan. Creating large numbers of game ais by learning behavior for cooperating units. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8, 2013.

[66] Georgios N. Yannakakis and Julian Togelius. *Generating Content*, pages 151–202. Springer International Publishing, Cham, 2018.

Declaration of Authorship

I hereby declare that this thesis was created by me and me alone using only the stated sources and tools.

Christian Wustrau Magdeburg,