Simulation-Based Believable Procedurally Generated Video-Game Villages

L.M. Clappers

Master Thesis DKE-21-50

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science of Artificial Intelligence at the Department of Data Science and Knowledge Engineering of the Maastricht University

Thesis Committee:
Dr. M.J.B. Stephenson
Prof. dr. M.H.M. Winands

Maastricht University
Department of Data Science and Knowledge Engineering
Maastricht, The Netherlands

August 23, 2021

Abstract

This master thesis explores a new method for procedurally generating villages in videogames. Designing villages is a time-consuming and expensive aspect of videogame design. Through procedurally generating villages, some of this burden could be taken away from designers. The implementation described in this thesis uses needs-based AI to simulate the needs of a growing village, applied to the videogame Minecraft. At every tick, the different needs of the village (for example food or housing), get decreased by a specified amount. Every type of structure satisfies the needs of the village differently. This leads to a generator that produces villages in a similar way to how they would develop in the real world. This kind of generator has many variables, that are initially set to values that intuitively make sense. Using an evolutionary algorithm, the values of these variables are tuned to fit different fitness functions, in order to become a more believable generator. The variables tuned are concerned with two aspects of placing buildings: which buildings get placed when, and where these buildings get placed. There are four different fitness functions, and five resulting types of villages: the Historical fitness function tries to optimize for a village that is most historically accurate, and the Balanced fitness function uses the ideas from the Historical function, but uses them in a less rigid manner. The Spread- and Need-based fitness functions focus solely on where to place a structure or what structure to place, respectively, and the Random generator places buildings completely randomly. For all four fitness functions a generator is trained, and for all five generator types a village is generated in three different biomes. A survey is performed on the results of these generators. All participants in the survey are shown short clips of the different villages, and asked about the immersion, fun, and realism that they would experience if they were exploring these villages in Minecraft. These three factors together were argued to be a good indicator of believability. The results of this survey indicate that the Balanced generator is capable of producing believable villages, but that this is also biome-dependent. Furthermore, it did show that the Historical generator did not perform satisfyingly, since this generator under-performed in every biome. This indicates that "true realism" is not necessary for making gamers experience realism.

Contents

Chapter 1

Introduction

This chapter is the introduction to this thesis about procedurally generating villages for videogames. First, the concepts that are relevant to this thesis are explained, then the motivation for this research is discussed, after that the problem statement and the three research questions are made explicit, and finally the outline of this thesis is given.

1.1 Concepts

In this section, concepts that are relevant for this thesis are explained. Those concepts are: procedural content generation, specifically for videogames, believability in videogames, and what is meant by a village in this thesis.

1.1.1 Procedural Content Generation for videogames

Procedural content generation (PCG) can be defined as the creation of game content with little to no input from a human [1]. Game content here can mean a range of things one would encounter in a game: from story to textures, from entire levels to characters. PCG can be used for a variety of reasons: to create more content than would be achievable by human designers, to enhance the replayability of a game, to make a game adapt to the player, or to make a type of game that would not be possible without PCG.

1.1.2 Believability

In this thesis, the concept of believability is fundamental, but this concept can be difficult to define. Broadly speaking, in this thesis when the believability of content is discussed, what will be meant is it is easy to believe that the thing you are seeing would exist in that world. It can also be defined as making it easy to suspend disbelief: if there are no jarring reminders that this is not the real world, then the content might be considered more believable. Believability is important because it contributes to a feeling of presence or immersion: when gamers feel "inside" the game [2].

Believablity does not have to go together with realism. Often what humans consider to be believable is not the same as what would be realistic. Many games for example fiddle with percentage chance that are displayed to the player, because showing the player accurate chances would make them feel it is unfair. A player who does not hit a 33% percent chance 3 times in a row would feel like the game is cheating, so game designers often choose to use different internal chance calculations than those that are presented to the player [3]. Furthermore, sometimes realism does not work within the fiction of the game in question. Having your player character need to use the bathroom, for example, would be distracting and would take you out of the game. Experienced realism however does seem to go well with believability. If what is seen in a game feels realistic (even though it might not be completely true to life) it will enhance how easy it is to believe the scenario presented.

1.1.3 Village

Since this thesis is about generating villages, it is useful to define the term village as used throughout this thesis. The Cambridge dictionary defines village as "a group of houses, stores, and other buildings which is smaller than a town". This is largely the same as how this term will be used in this thesis: a collection of buildings and structures, meant to invoke a feeling of people living there. In videogames, an important aspect of villages is the purpose they serve for the people playing the game. This can be an aesthetic purpose, or villages can serve to make a virtual world feel more real or believable, there can be game mechanics linked to the location of the village, or any of a number of other functions. In this thesis, the focus will be on the believability aspect of videogame villages: how they make the world feel more believable.

1.2 Motivation

As videogame worlds become more and more expansive, more content needs to be built for these worlds. Building this content by hand is a task that is work-intensive and expensive. Therefore, being able to leave this task to a computer would make developing large videogames easier, cheaper, and faster. Furthermore, there are videogames in which important aspects of gameplay are based around interacting with content that changes each playthrough: rogue-likes, for example, have the player replaying the same game over and over again, while shuffling their content, and Minecraft's world is randomly generated each time you start a new game. For these types of games, making content that seems more like careful design has gone into it and less like the result of a generating program, helps with immersion in the game and therefore with enjoyment of the game. Building a procedural village generator that produces believable content, therefore, will help with immersion while playing videogames, and will make building large open world environments less labor intensive.

For this thesis, a simulation-based approach was chosen: more specifically, simulating the village as an entity with needs. This was chosen because this approach has not been used before for generating this type of content, and is therefore a novel approach to the problem. It is also a quite promising idea: villages in the natural world also develop over time, with new buildings built as the village develops a need for them. Because this approach seemed grounded in reality, the hope is that it would help with believability. Another advantage of this simulation-based method, is that the villages could change dynamically over time. Since the simulation does not have a definite endpoint, this system could be used to believably change and expand the village every time the player returns to it, thereby giving the idea of a world independent of the player, and therefore increasing immersion.

It was chosen to develop this village generator for the game Minecraft specifically [4]. This was done because of multiple reasons: since Minecraft has a large modding community and has existed for a long time, many resources exist that allow for easy code-based changes to the world. Because of this easy modification of the worlds, there is also much earlier research done in Minecraft, specifically for PCG. There is even a contest for procedural village generation in Minecraft, Generative Design in Minecraft (GDMC) [5]. Furthermore, because of the many available modding tools, it would be relatively easy for others to use the code made for their own purposes, and make improvements.

1.3 Problem Statement and Research Questions

In this thesis, the aim is to find a new solution to procedurally generating villages for videogames, focusing on believability as a goal. The problem statement is the following: "How to develop a procedural village generator that produces (more) believable villages for Minecraft-like games?" To address the problem statement the following three research questions are formulated:

- What existing ways of procedurally generating villages for videogames are there?
- How to build a simulation-based village generator for Minecraft-like games?
- How to optimize the parameters for this village generator for believability?

These research questions were chosen because together they span most of the relevant investigation into procedurally generating villages for Minecraft-like games. The first research question is concerned with earlier research. This is necessary to answer the problem statement satisfyingly, since in order to be able to conceive of different solutions, one first has to know what solutions were already tried. The second research question asks to build a simulation-based generator. This contributes to the goal of building a believable generator, since it is believed that a simulation-based generator would be more believable. The final research question is about optimizing this built generator for believability, so its variables are not just set to random values, but to values that optimize believability.

1.4 Thesis Outline

The outline of the thesis is discussed briefly. In Chapter 2, some relevant concepts in procedural content generation are discussed: search-based techniques, mixed initiative, and how to evaluate procedurally generated content. In this chapter earlier research on terrain generation, village generation and need-based AI are also discussed. In Chapter 3 the Generative Design in Minecraft competition will be explained, and some earlier submissions to this competition are shown. Chapter 4 describes the methods used for this research. It is split into two sections: a part about the algorithm of the simulation-based village generator, and a part about the search-based approach that helps to tune the generator. After that, in Chapter 5, the experiments executed to train the generator and to validate the trained generator are described, and the results are given. In Chapter 6 the conclusion and discussion is given, and future work and improvements are discussed in Chapter 7.

Chapter 2

Background and Related Works

In this chapter background information will be given, and some earlier related works will be discussed. First, the concept of procedural content generation, as well as some subjects within this field relevant for this thesis will be explained. After that, related work will be discussed, which will focus on terrain generation, village generation, and simulation of needs.

2.1 Procedural Content Generation

As stated in the introduction, procedural content generation (PCG) for videogames defined as generating content with little to no input from a human designer [1]. Here, three subjects within the field of PCG will be described, based on what information is relevant to this thesis: search-based PCG, the concept of mixed initiative in PCG, and how to evaluate procedurally generated content.

2.1.1 Search-based Procedural Content Generation

The fundamental idea of search-based procedural content generation, is that there is some form of algorithm that is searching for content with the desired properties by adjusting parameters. It is exploring a search space filled with possible solutions, and trying to find the best or a satisfying solution within this space. Often, evolutionary algorithms are used for these kinds of approaches, but there are other options [6].

A search-based approach has three core components: the search algorithm, the content representation, and one or more evaluation functions. As stated before, for the search algorithm often an evolutionary algorithm is chosen. It uses randomness, combining the properties of well-functioning individuals and picking the best individuals to increase the performance of the population. The content representation is how the content is presented to the search algorithm in a way that it can be easily changed and optimized. The evaluation function tries to measure the quality that is being improved, whether that is fun, interest, believability, or something completely different. It assigns some value to the content that has been generated that makes it possible to rank the different results among each other, so the search algorithm is able to optimize for well-performing content [6].

Evolutionary Algorithms

Evolutionary algorithms are one of the most used form of search-based algorithms [7] [8] [9]. They are inspired by the concept of natural selection. The content representation of these evolutionary algorithms is usually referred to as a gene, and the content this gene applies to is an individual. All individuals together make up the population. Every time the algorithm is ran, this is referred to as a generation, and every generation the entire population gets replaced by a new generation. There are multiple ways of making the new generation. The first way is to replace the worst λ individuals with copies of the others, and then mutate these λ individuals, to allow for randomness. This is called the evolution strategy. Another type of the evolutionary algorithm is a genetic algorithm. In this algorithm the genes of two individuals from the older generation can be combined into one or two new individuals (mating these individuals). The new individuals are called the children, and the individuals the genes came from are the parents. When genes are combined, this can be done in a variety of ways. The most conventional one is a form of crossover. The genes are then combined by taking the first part of one gene, and the second part of the other gene, as can be seen in Figure 2.1. After crossover has occurred, there is usually some form of mutation that can be done. There are multiple possibilities for this as well. The simplest form of mutation is point mutation: one variable on the gene gets changed by a certain amount. Depending on the form of the gene, mutations could be more complex; if the gene is presented in a graph form, a mutation might switch a part of the graph out with a predetermined other partial graph. Often in genetic algorithms, some form of elitism is applied. With elitism is meant that some of the best individuals from the parent population get put directly into the child population. This practice can make learning less likely to collapse, but it can also harm the variability of the population.

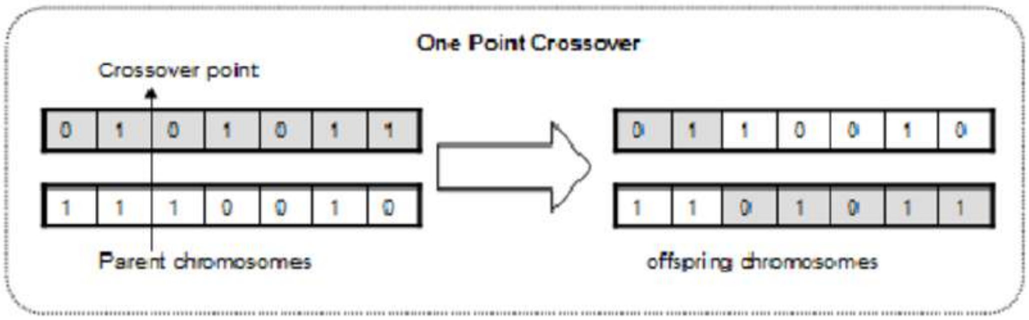


Figure 2.1: *Visualization of crossover [10]*

Since the evolutionary algorithm that is used in this thesis is a genetic algorithm, this type of evolutionary algorithm will be focused on [11]. Algorithm 1 shows an example of a genetic algorithm.

There are different ways to select the parents from the population. The one discussed here is the tournament method, which is the parent selection method used in this thesis. It works as follows: from the population, k individuals are selected (with or without replacement), and the best (two) individuals from this group are selected to be parents. Depending on how large k is, this can be more or less selective pressure: the larger k is, the more likely it is that the best individual in the tournament is the best individual overall, while a k of 1 is the same as random selection.

Algorithm 1: A Genetic Algorithm

```
initialize the population;
set generations to 0;
evaluate the population;
while generations < maxGenerations do
    set childPopulation to emptyList;
    if elitism then
        add the best individual(s) of population to childPopulation
    end
    while size(childPopulation) < size(population) do
        pick parent1 and parent2 from population;
        mate parent1 and parent2 to get child;
        if randomValue < mutationChance then
            mutate child
        end
        add child to childPopulation;
    end
    set population to childPopulation;
    evaluate the population;
end
```

2.1.2 Mixed Initiative

The term mixed initiative is used to describe a system where both human and computer creativity is used to generate content. There are different ways in which this can be done: computer aided design, in which the human designs most of the content with the computer as a tool to make content creation easier, and interactive evolution, in which the computer creates the content, but humans give feedback to the computer to make it fit their preferences [12].

Computer aided design comprises a range of possible solutions, from level editors that allow human designers to place the entities of a level, to programs that take simple drawings and make them into an entire 3D environment, to generators that take as input only a few settings. It is used to make generating content faster and less expensive, while still allowing for (an amount of) influence and creative control from the human designer.

Interactive evolution is when the fitness evaluation of an evolutionary algorithm is done by having humans to judge the content. Many criteria that are deemed important when generating content, such as fun, believability, immersion or challenge, are difficult for a computer to objectively measure. They are almost by definition subjective criteria that are not easily measurable from the raw data of the level, even if most people would agree on which levels are more fun than others. Therefore, often these sorts of criteria are not evaluated by an automatic fitness function, but are judged by humans, who are shown a piece of content and have to assign it a value. The advantage of this approach is that it is more true to what is tried to be measured, but it also has its disadvantages. Humans can get tired, and they judge more slowly than computers. It is therefore quite expensive to get humans to serve as a fitness function. Furthermore, humans find it difficult to judge experienced difference if there is not a large contrast in the content, and they display more variability in how they judge. For these reasons, when an interactive evolutionary solution is chosen, it is often mixed with more conventional fitness functions.

2.1.3 Evaluating Content

Being able to evaluate the results of a generator is important. Building a generator is easier than building a good generator that generates the type of content the designer wants. Many of the features generators are designed to build content for are subjective qualities, and therefore difficult to measure. There are two main ways of evaluating a generator: the top-down approach, which uses generator data to visualize the type of content the generator makes, and bottom-up evaluation via players, which uses questionnaires to measure human subjective opinions about a game [13].

The top-down approach is concerned with the expressivity of the content created. Checking the expressivity can make sure that the generator performs well in multiple circumstances, not just in the few testcases that can be checked, and that the generator provides a wide enough range of content. The principle is that a large amount of content is generated and evaluated, and that the result of these evaluations is then visualized, with for example a heat map or a histogram. What metrics the content is evaluated on deserves some attention: best practice is to make these metrics be as far as possible from the input to the generator, since it would otherwise just measure whether the generator is performing well, instead of whether it has the kind of expressive range aimed for.

The bottom-up approach makes humans experience the content, and then asks them for their opinion. This is usually done through surveys, especially ones that ask the people to rank several options. Ranking is preferable over rating, since it gets rid of an amount of biases, and makes it more likely that the different participants give similar answers [14]. Instead of surveys, other types of information can also be extracted by human experience. Examining the play-style of gamers, what levels they get stuck on and where, what they look at in a level, what aspects hold their attention for the longest amount of time, can be quite informative depending on the quality that is being measured.

2.2 Related work

In this section, earlier research that is similar to the work done for this thesis is described. First, some interesting earlier approaches to terrain generation are shown. After that, earlier work in village generation is discussed, and finally some examples of needs-based AI are shown.

2.2.1 Terrain Generation

Terrain generation refers to automatically generating terrain that can usually be rendered as a 3D environment. This is related to using PCG for villages, and many of the village generators similar to the one described in this thesis use some form of terrain generation. This technique is only applied in this thesis in a very limited way: parts of the map are flattened.

In this section, multiple different examples of terrain generation are described. In [15], different techniques for terrain generation using noise, simulated erosion, and simulated vegetation are described. The methods are compared to each other, and described in terms of speed, memory usage, and quality. An application of some of these techniques can be seen in [16].

A different approach to terrain generation is shown in [17]. Here, an evolutionary algorithm is applied: using search-based techniques to find good terrain. Different mathematical and noise functions are blended in such a way to achieve a satisfyingly mountainous terrain, with enough accessible areas, and and enough edges in the terrain to maintain interest.

It is also possible to use agents in terrain generation, as Doran and Parberry [18] described. They have multiple different types of agents walking around on a map, building different features as they go along. Each different type of agent has a different set of rules: coastline agents, for example, can create land by elevating land that is under sea-level, and the river agent moves uphill from a coastline point to a mountain point until the mountain is reached, and then moves downhill again to build its river. Another feature of this method is that it incorporates some mixed-initiative techniques: a designer can set certain parameters for the different agents, and can influence the world after it has been built.

2.2.2 Village Generation

Villages or cities can be generated on different levels of abstraction. Do you also generate entire houses, or just a flat map with zoning designations? Or something in between? Different approaches to village and/or city generation are described in this section.

A prevalent approach to generating cities seemed to be to make some large 2D map, such as a city plan. Different approaches can be taken to this goal. In [19], different agents develop a terrain map into a city map by following their own rules that are dependent on the type of agent they are, similar to [18]. Vanegas et al. [20] take another approach: there the modeling of the geography (where roads, parcels and buildings are) and the agents (who determine the amount of jobs, population and land value) come together to make a 3D model of urban space.

Village generators that operate on a smaller scale are often less complex algorithms with emergent features. Early city generators, such as [21] used L-systems (a type of PCG where a set of rules can be used to generate tree-like shapes [1]). Different L-systems were used for building the roads, and for the clusters of buildings that form the villages. Sometimes, some relatively simple algorithms already gave good results: for example a village generator that was basically an A* road network which added buildings to the sides of the road [22]. A more recent algorithm uses cellular automata to generate cities [23].

2.2.3 Simulation of Needs

In this section, earlier examples of need simulations of an individual or entity will be described, since simulating an entity is a relevant idea for this thesis. First, the main inspiration for the simulation method, the need-based AI as seen in the videogame "the Sims" will be discussed. After that, some applications of these need-based systems on human behavior modeling are shown.

In [24] the concept of needs-based AI is explained. It describes how the AI of individual sims in the game The Sims works, and how that seems to give sims agency and makes them behave in believable ways. The basic principle is that there is an action queue, with actions the sim is planning to take. Every sim has needs, and each needs depletes in a different way. The objects around the sim advertise satisfaction of needs to it, and different objects satisfy different needs in different amounts. Before putting an action on its queue, all available nearby options for satisfying needs are examined, and one of the three options that satisfy the sim's need the best is randomly chosen. This system is inexpensive, and provides behavior that is believable, in a sense. This algorithm has been used in multiple iterations of the Sims games, and in other simulation games [24].

Related to the concept of needs-based AI, is utility-based AI [25]. This design pattern for AI comes down to making a list of options, evaluating those options, and choosing one of these options to execute. Needs-based AI could be considered a more specific implementation of this design pattern, since the same principle is followed, but the evaluation method specifically has to do with the needs of the agent.

Aside from video-game AI, the needs-based concept has been used in models to explain human behavior. In [26], a need-based system is shown that is used in order to predict activities people will take, and through that predict travel. Nijland et al. [27] expand on this concept by implementing activities that satisfy multiple different needs at the same time.

Chapter 3

Generative Design in Minecraft (GDMC)

This chapter discusses the Generative Design in Minecraft competition (GDMC). First, the contest, its submission methods and criteria are described, with special attention to the Chronicle Challenge, as well as to the changes made to the contest in 2021. After that, some notable earlier submissions will be discussed.

3.1 Competition

The GDMC competition was first held in 2018. The goal of the competition was to advance PCG research in two particular ways: adaptivity to content, and holistic PCG [5]. Adaptive PCG was defined as content generation that would adapt to content that had already been generated. More specifically to Minecraft: GDMC was meant to stimulate generators that left the already existing world mostly intact, and that changed the generation results to fit with the environment, instead of the other way around. An algorithm that can generate a village on a completely flat map without any features is less interesting than an algorithm that can adapt to what interesting features there are on the map, such as mountains, rivers, or even earlier built structures. With holistic PCG is meant that the different aspects of the generator fit well together. It is meant to stimulate villages in which all aspects of the generator are meant to support the final result. This is done by grading the village as a whole, and not grading individual aspects. There are no grading criteria for small-scale content, such as the aesthetic value of the buildings specifically, but rather the aesthetic value of the village is graded in totality, for example.

Since 2018, the competition has been held four times, once per year. In the first year, only four generators were submitted [28]. Every year the number of submissions has increased, with this year's (2021) submissions numbering 20. The competition has developed over the years, adding new criteria, submission methods, and optional challenges.

3.1.1 Grading Criteria

Village generators submitted to the GDMC are judged based on four aspects. These aspects are:

- Adaptability
- Functionality

- Believable and Evocative Narrative
- Visual Aesthetics

All generators are judged on these aspects by a panel of judges, who grade the generator on how well it satisfies these criteria. Every generator is tested on a couple of different maps, with different biomes and elevation levels.

Adaptability judges what it name indicates: it is concerned with whether the generator produces villages that are adapted to the already existing map. This can be in terms of respecting the "natural" terrain differences, building the buildings out of materials that exist in the area, or even adjusting the shape of the buildings depending on what biome they are build in. It also includes co-creativity: building a village around (and possibly even in a similar style as) structures someone else has already built.

Functionality is concerned with what the person playing Minecraft might experience. The question it asks comes down to: "Does this village function well in the game?" This includes resources that the player might need, such as food and crafting stations, that every part of the village is reachable by the player, and that the village is safe enough to rest there during the night. This functionality criterium can also be considered from the view-point of the non-player characters (NPCs), the villagers who "live" in the village. Does it seem like they have what they need to survive in this village?

The next criterium, narrative, is concerned with a believable narrative. Can you imagine the a story/culture/history for the generated village? Do different villages made with the same generator serve different functions? This criterium is historically the one that the least attention is paid to by the competitors [29]. Possibly, because actually building a village with a complex history is more complicated than building a beautiful city that is well-adapted to its surroundings and that functions well: the history affects all other aspects, and depending on what happened in the village history the building types, materials used, layout of the city, etc. can be affected throughout the entire village.

Visual aesthetics is concerned with whether the city looks good. While of course taste is different for everyone, there are definite differences between villages that are more and less good-looking. Interesting architecture that is more than just a square house will be judged more favorably. Another aspect of visual aesthetics, is that there are no obvious artifacts of the generator, since this would be considered to be ugly by most judges.

3.1.2 Chronicle Challenge

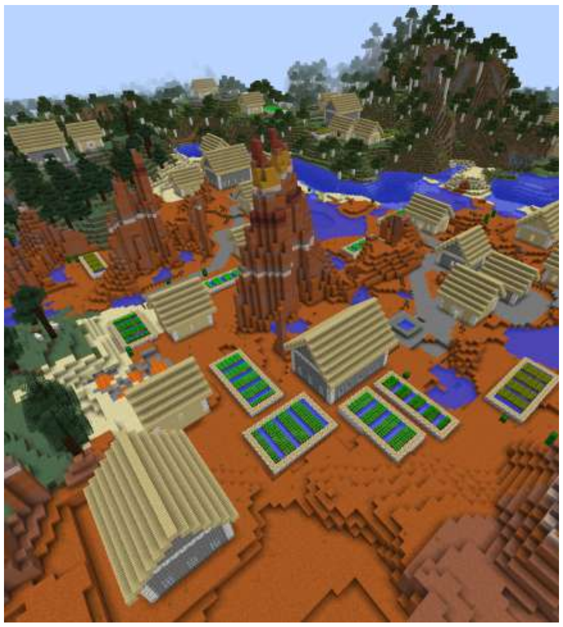
The Chronicle Challenge was introduced in 2020. The challenge is to provide a chronicle somewhere in the village that explains the history of the city, in order to give the village more of a narrative [29]. This can be done after the village is generated, by some model that makes up a narrative to explain the features that result from the generator, or it can be taken into account during the generation process and use some of the same logic to explain the village. It can also be a complex computer generated story, treated as a natural language problem, or it can be solely preconstructed sentences where words are replaced depending on the village that was generated.

3.1.3 Changes in 2021

There were some changes and additions made to the competition in 2021. Two new non-optional challenges were added: a 1000 by 1000 map (around 16 times larger than the normal map of 256 by 256), and co-creativity. For the 1000 by 1000 map, the challenge was to find the best place within this very large map to actually develop the settlement. Then, there would have to be some form of guidance towards the judges from the center of the map to the village (in the form of a road, an arrow, or a signpost, for example). The challenge with co-creativity was to incorporate already built structures into the generated village in some form: from building around them and leaving them intact, to copying the style of the earlier built buildings. The final change in 2021 was that instead of all maps having the same set size of 256 by 256, the size of the maps on both the x and z direction would range between 200 and 300. This meant that any generator developed would have to be able to deal with differently sized maps.

3.2 Earlier Submissions

In this section, some earlier submissions to GDMC will be shown. Interesting submissions each year will be discussed, with the most attention being given to submissions with more extensive explanations provided and innovative approaches.


(a) *Submission in 2018*

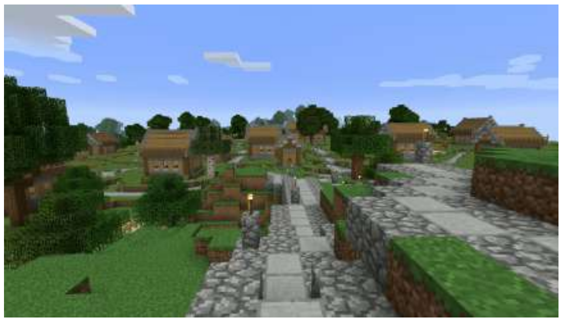

(b) *Submission in 2019*

Figure 3.1: *Examples of the results of Filip Skwarski's submissions in 2018 and 2019*

In 2018, the first year that GDMC was held, Filip Skwarski won the competition. His approach used a scoring system to evaluate possible locations for a new structure, and then place one of two types of structures on that location. He only used two types of buildings: plazas and farms. He generated roads with an $A*$ algorithm. The resulting villages have clusters of structures close together. The next year, in 2019, Skwarski improved on his earlier submission by adding more different buildings, and by having his already existing buildings "suggest" what buildings should come next to them. With this new and improved algorithm, he won the competition again. Images of the results of these two generators can be seen in Figure 3.1 [30].

The second place in 2019 went to Julos14 (Figure 3.2), a team of bachelor students from Denmark. Their approach was different: instead of picking building locations first, and letting the layout of the entire village emerge, this team decided to divide the map into sections separated by

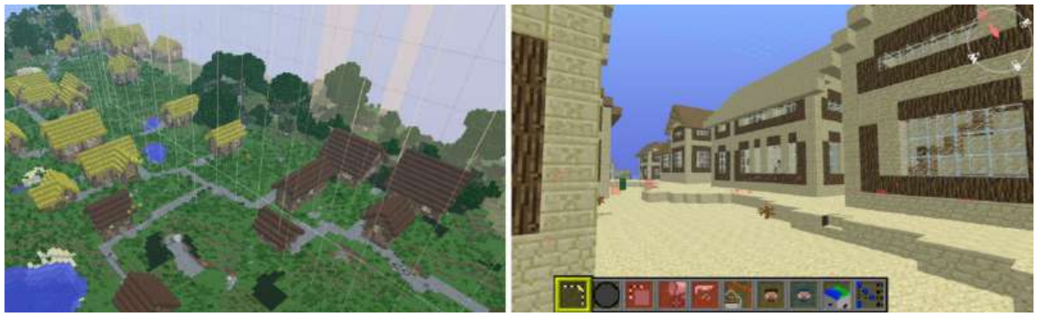


Figure 3.2: *Examples of the result of Julos14 [31]*

hills or water, and build a network of roads within each section, after which building locations get assigned [31]. The fact that this team got second place with an opposite approach from Skwarski shows that no optimal approach to this problem had been established.

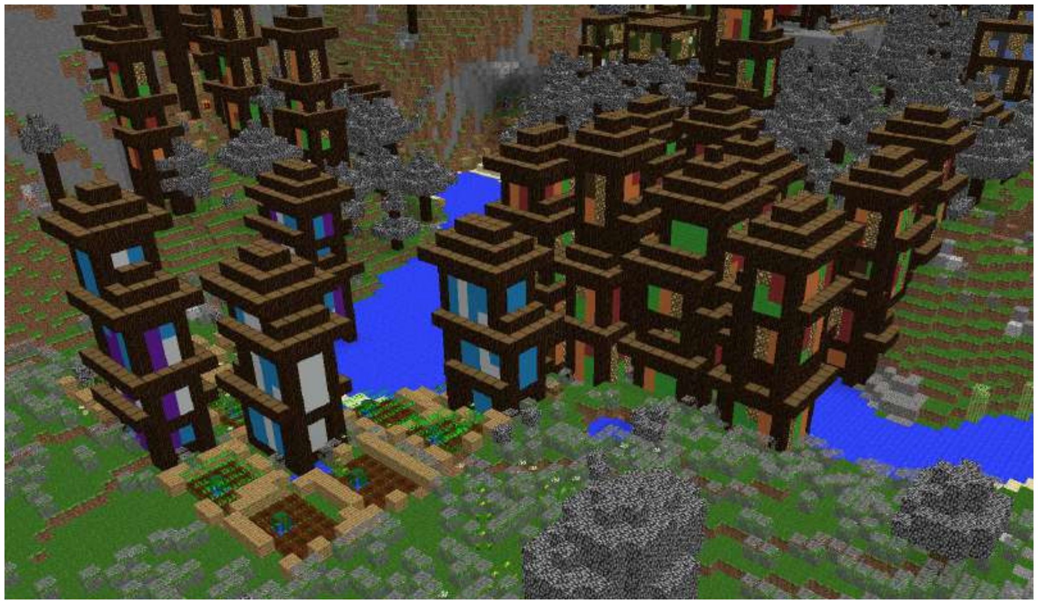


Figure 3.3: *Examples of the result of the World Foundry [32]*

In the 2020 competition, a number of interesting generators were submitted. Three are highlighted; The World Foundry, ICE_JIT, and the University of Tsukuba. The World Foundry used a simulation based approach. Autonomous agents walked around on the map, and placed buildings with the materials at that spot, using their own style. When two agents came together, they had a chance to create a child that inherits style aspects from both parents. What kinds of buildings were placed depended on features in the landscape. This generator was the first to take on the Chronicle challenge: every agent keeps a log of their actions, which is then placed in the world as a chronicle [32]. The method the World Foundry used for placing the chronicle in the world has been adapted in the generator made for this thesis.

ICE_JIT based its style on Japanese buildings. Their villages have different types of Japanese styles in different biomes, with mountainous biomes having more pagodas and shrines, dessert

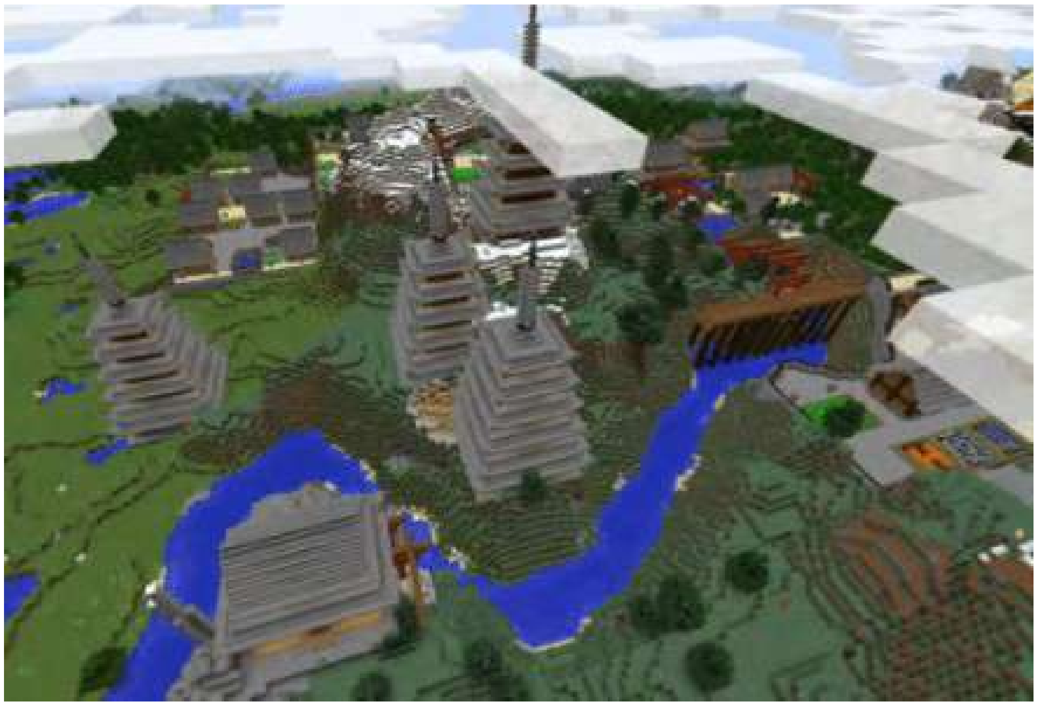


Figure 3.4: *Examples of the result of ICE_JIT [33]*

biomes having Japanese townscapes, and regular terrain being more reminiscent of Kyoto. The generator generates a main street, with smaller side streets branching off it, on which houses and shops are placed [33]. The results of this generator can be seen in Figure 3.4.

The final generator to be discussed here is the Traincity generator, by the University of Tsukuba, which is shown in Figure 3.5. It is based on an earlier submission, by ehauckdo in 2019. This submission focused on dividing the map into a city center, and the outskirts of the village. In the city center, apartment buildings were placed, while in the outskirts smaller houses are build. The addition made by the University of Tsukuba is that there is a train-network, that allows villagers to be transported from one side of the city to another [30]. The different structure generators from the University of Tsukuba were used in this research to generate almost all structures seen in the generator.

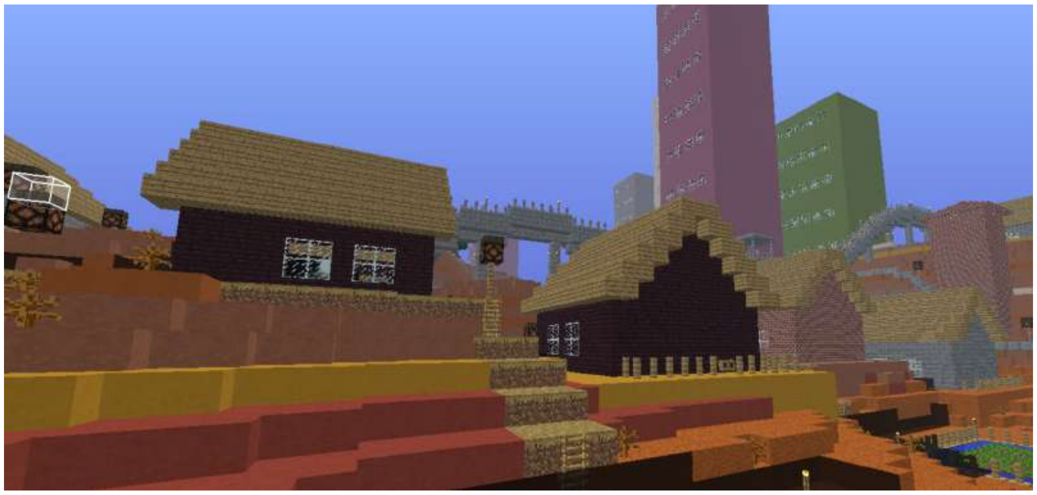


Figure 3.5: *Examples of the result of Traincity*

Chapter 4

Building the Generator and Optimization

In this chapter, the methods to build the village generator are discussed. First, the general set-up of the simulation-based village generator is discussed, then the optimization of the parameters of this generator is described.

4.1 Simulation-based village generation

In this section, the process of creating the simulation-based village generator is discussed. First, the framework that was used (MCEdit) will be explained briefly. After that, the algorithm that was developed is explained in depth, with a particular focus on the initial location finding algorithm, the needs calculation, and the location and costs.

4.1.1 MCEdit

MCEdit is a "saved game editor" for Minecraft. Because of the way that Minecraft save files include the entire generated world, this means that it is similar to a world-editor program. It is not affiliated with any of the official distributors of Minecraft, such as Minecraft, Mojang AB, or Microsoft Inc., and is an open-source project [34]. MCEdit provides certain utilities that make it a very convenient method of changing the Minecraft maps in any way that is desired. For the purposes of this research, the filter function is especially useful. It allows for easy editing of large chunks of the map through scripts written in Python, which makes writing a PCG algorithm for Minecraft much easier. The version of MCEdit that GDMC provides also comes with an amount of functions and methods that provide utility for generating villages specifically.

4.1.2 Simulation algorithm

The main concept of this approach to generating Minecraft villages is to simulate the entire village as an entity with needs and wants. The village can then fulfill its needs through building different houses that satisfy different needs in different ways. It is based on the needs-based AI system that is used in the Sims [24]. Every action that can be taken is weighed for satisfying the sim's needs the best, and one of the top three actions is chosen. Inspired by this approach the algorithm for this village builder was made. It is shown in Algorithm 2.

Algorithm 2: The need-based villagebuilding algorithm

```
1  find the best starting spot;
2  build the first building there;
3  set ticks to 0;
4  while ticks < maxTicks do
5      update needs;
6      for buildingType in allBuildingTypes do
7          set costs to $\infty$;
8          for buildingLocation in allPossibleBuildingLocations do
9              if cost for buildingLocation < costs then
10                 set costs to cost for buildingLocation;
               end
           end
11         calculate need improvement for buildingType;
12         calculate score using costs and need improvement ;
       end
13     determine best score;
14     if best score > 0 then
15         build best building
       end
   end
```

In this section, multiple parts of this algorithm are discussed in turn. First, the different input variables to the generator is shown. After that, the initial location selection procedure is discussed. Then, the needs calculation is explained. After that, the location selection and cost calculation is discussed, and finally some small additional features of the generator are explained.

Input parameters

There are a number of input parameters to the generator that have to be discussed first. Some are explained in this section, others fit better in another section, and are explained there.

- The first input variable to be discussed here is the max number of buildings. If this is set to zero, there is no maximum, but if it is set to a value larger than zero, the generator will stop generating buildings after that maximum is reached.
- Simulation ticks is the amount of ticks the generator is allowed to run for.
- Bridges is a boolean that determines whether the bridge building algorithm should be turned on. The bridge building algorithm does not function well, and therefore this boolean is off by default.
- Clear trees determines whether the generator clears the trees in every new area that is added to the area where it can search for new building locations. This is by default set to `false`, because it is quite expensive to check every square of the new area for trees. Especially when training these generators, the costs need to be kept as low as possible to finish in a reasonable time.
- Co-creativity avoids building on top of earlier built structures, but has not been tested intensely, and is expensive since every square in the new area has to be checked for pre-built blocks.

Parameter name	Type	Range	Standard Value	Explained
Max number buildings	int	[0, ->)	0	here
Simulation ticks	int	[0, 50]	30	here
Search area	int	[16, 30]	20	Initial Location Selection
Water	float	[0.0, 1.0]	0.7	Initial Location Selection
Lava	float	[0.0, 1.0]	0.2	Initial Location Selection
Cliffs	float	[0.0, 1.0]	0.1	Initial Location Selection
Scoot	Boolean	True, False	True	Initial Location Selection
Bridges	Boolean	True, False	False	here
City wall	Boolean	True, False	True	Smaller Aspects
Chronicle	Boolean	True, False	False	Smaller Aspects
Clear trees	Boolean	True, False	False	here
Co-creativity	Boolean	True, False	False	here
EA version	Boolean	True, False	False	here
Load genes	Boolean	True, False	False	here
Version	String	Random, Spread, Needs, Historical, Balanced	Random	here

Table 4.1: *Input parameters*

- EA version turns certain expensive parts of the code that are not necessary for learning off, such as finding the initial location.
- Load genes allows the algorithm to load the variables that were determined by the evolutionary algorithm for different fitness functions, and version determines which genes will be loaded, and whether location or needs-based calculations will be made, depending on the version chosen.

Initial Location Selection

The concept behind the way in which the initial location was selected, was based on where a first settler would be likely to build their first building. Most villages are built close to water, so water was one of the things this algorithm would be searching for. Other aspects that would be searched for would be cliffs, because settling near a cliff would provide security, and lava, because it would be convenient for players to have a village near lava.

In order to find this first spot, the entire map was divided into sections of 16 by 16 squares. The outer-most squares were not considered to place the first building into. For each square, it was checked whether it only existed of valid ground blocks (blocks that are not water or lava). If it did, it was added to the pool of possible squares to place the first building on. After that, every square was checked to see whether it had one or more neighbors that were a square containing a large amount of water, lava or a cliff. If the square did have neighbors containing these features, it was added to the respective list of squares neighboring one of those features. One of these squares would get selected, biased by user input and by centrality in the map. If "scoot" is turned on, then a slightly wider square gets considered, and within that square the flattest area that fits the starting structure that is closest to the features will be selected. This procedure corresponds with line 1 (and 2) of Algorithm 2.

Needs Calculation

After every tick, the needs of the city get updated. This corresponds to line 5 in Algorithm 2. There are five different needs in total: food, housing, jobs, resources, and social. These needs were chosen because they fit well with the structures from earlier work that were used in the generator, and because they seemed to be some of the factors that would be important for a developing city. However, many other needs could have been included, such as security, or transportation. For more on this, see Chapter 7.

For the first 15 ticks, only the food and housing needs came into play, after that all needs were active. Every tick, the population increased, either by a growth factor (used during training to decrease unnecessary variability) or with a certain chance to add one individual. After that, every relevant need was updated by subtracting the population minus the amount of satisfaction of that need already in the village, bounded between 0 and 100.

$$Need = min(100, max(0, (OldNeed - (population - satisfaction)))$$

The satisfaction for each need is depended on the buildings that are already placed. Every type of building satisfies one or more needs, and the amount by which the satisfaction increases for that building is determined by the genetic algorithm. So, for example: if a house satisfies housing by 3, and there are 2 houses already built, then the satisfaction for houses is 6 ($3 \times 2 = 6$). If the population then is 7, and the old value of the need is 46, the new value will be 45: $46 - (7 - 6) = 45$.

Location and Costs

To select the next location for a structure to be built, an area containing the entire village up to that point, plus a strip of land around the borders of the size of the variable "search area" is investigated to find possible new locations for a structure. Every spot that was not already occupied or contained water is considered, and for each different building type seven possible building spots were selected (the number of building locations investigated was limited to seven, because that seemed to provide a good balance between relatively quick finishing, and a satisfying structure location). The spot with the lowest cost, as determined by the unevenness of the location, is selected to be the building location for that type of building. This corresponds to lines 7 to 10 in Algorithm 2. The different building types, their sizes and the needs they satisfy can be seen in Table 4.2. Many of the buildings are aesthetically only a variant of the "house" building, with a different usable object inside. Whether this is the case for any of these building types will also be shown in Table 4.2.

Selecting the Next Building

After the locations for the next buildings as well as the costs for these locations were determined, every building type is tested for how well it would satisfy the needs of the village (line 11 in Algorithm 2). Whenever a building is build, the need(s) it satisfies is given a bonus, the size of which can be set. Depending on how much the needs change, the action of building that structure is given a score. The formula for this score was taken from "Needs-based AI" [24], and it looks as follows:

Building name	Minimum size	Random range	Needs satisfied	House-like
House	14	6	Housing	n.a.
Farm	14	6	Food	no
Fountain	18	0	Social	no
Pasture	12	8	Food, Resources	no
Cleric	14	6	Jobs, Resources	yes
Crafter	14	6	Jobs, Resources	yes
Smith	14	6	Jobs, Resources	yes
Enchanter	14	6	Jobs, Resources	yes
Blacksmith	14	6	Jobs, Resources	yes

Table 4.2: *Different building types*

$$score = \sum_{allNeeds} (A_{need}(currentValue_{need}) - (A_{need}(futureValue_{need}))$$

with A_{need} being an attenuation function, in this case $10/need$. By using this function, fulfilling needs that are almost depleted becomes much more important than needs that are still almost fully satisfied.

This need fulfillment score is then balanced with the costs of building that structure at that location, resulting in a total score. The building type that has the best total score will be built, and its bonus will be applied to the needs (lines 12 to 15 in Algorithm 2).

Minor Additions

Some other small additions were made to the generator. The ones that will be discussed here are the chronicle system and the village wall generator.

Since this approach to village generation seemed well-suited to the chronicle challenge, a simple way of solving that challenge was implemented. Any structure that in the fiction of the village the villagers built got a string assigned explained what was done, and why this was done. The first building location, for example, would be specified as "The tower stands at the place our first settler started the city. This spot was chosen because it was close to water." and building a house would be described as "We build a house because it was good for housing." After the generator was done, the chronicle was converted into a book, and placed into a chest in the tower.

After a certain number of ticks, there is a chance for a village wall to be built. The main purpose of this wall was to make it seem like the village took security into account, and to show how the village had developed over time. After the wall was built, new buildings would still be added, and just like with medieval European cities, some newer buildings would have to be built outside of the village wall. Determining where the wall would come was relatively easy: it would be built just outside of the borders of the village at that tick.

4.2 Search-based Approach

In this section, the way in which the parameters of the simulation-based village generation algorithm are optimized is described. During the first iteration of this algorithm, the values for the amount of satisfaction a existing structure gives to a need, the bonus given to the agent for building any particular structure, and the search range were all set to certain values. These values

Gene type	Min	Max	Change
Satisfaction per building of this type	0	10	5
Bonus per building of this type	0	100	50
Search area	10	40	15

Table 4.3: *Minimum and maximum value, and the amount by which the gene maximally changes per mutation for different types of genes*

made intuitive sense, but were not tested at that point. With the evolutionary algorithm these variables are optimized to provide a result specified by a fitness function, which should provide better results.

In order to optimize the different variables for the best village-generation, an evolutionary algorithm (EA) was used. This method was chosen, because there were many independent variables that would influence the outcome, and therefore a more conventional hill-climbing algorithm would not have been a good fit.

In this section, first the content representation used for the evolutionary algorithm will be described. After that, the way in which selection is performed is discussed. The last and largest part of this section is an explanation of the different evaluation functions that were used for this research.

4.2.1 Content Representation

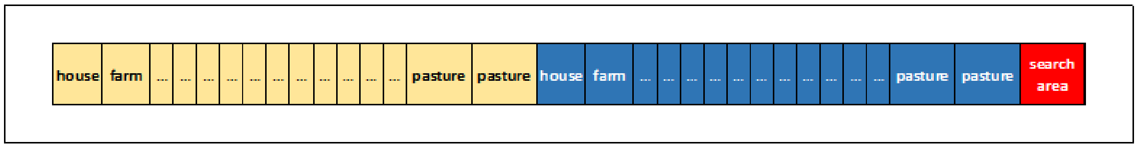


Figure 4.1: *Visualization of a gene*

Each individual has 31 genes. Of these, the first 15 code for the long-term need-satisfaction of the different types of buildings, the next 15 code for the short-term the bonus to the need for a built structure, and the last one codes for the distance at which new buildings can be placed (the search area). When the genes mutate, they can change a certain amount per mutation, and all positions are bounded. For more specific info, see Table 4.3.

4.2.2 Selection

In order to generate the children, first elitism is applied: the best individual is added to the child population, and the worst individual of the old generation is discarded. Then, parents are selected using tournament selection. Every pair of parents produces one child through crossover. This crossover takes into account that positions 15 away on the gene count for the same information, so these are passed on together (for an visualization, see Figure 4.2. The red square stands for the position at which crossover occurs). After that, there is a chance for the child to mutate.

4.2.3 Evaluation Functions

The purpose of this evolutionary algorithm, is to optimize the believability of the generated villages. While there is no efficient way to directly optimize for believability, some approximation could be

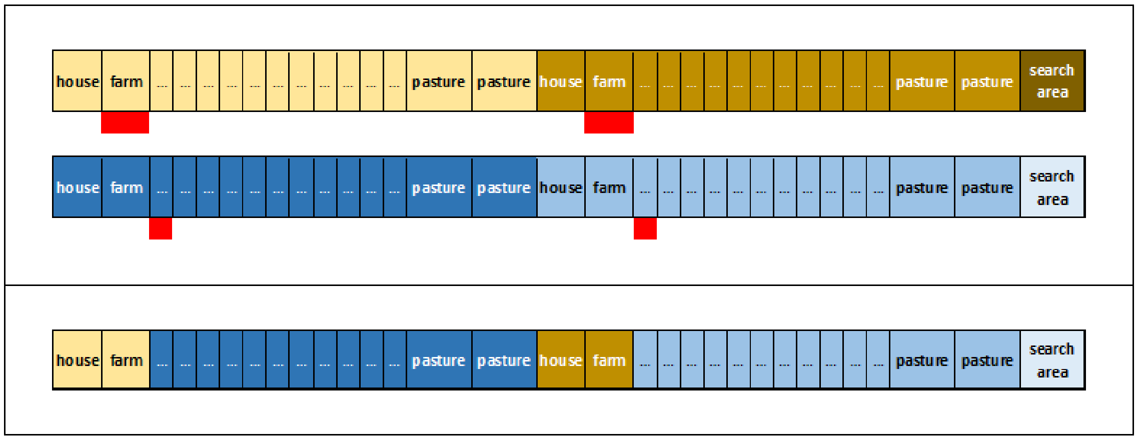


Figure 4.2: *Visualization of crossover*

done by taking inspiration from how this generator functions: simulating village development in the real world. Therefore, that is what the fitness function is optimizing for: criteria from literature on village development that are similar to how real villages historically developed. Based on this research, four different fitness functions (and a random generator) were developed. In order to test each fitness function, after evolving a village had to be generated for each individual in the population. After a village was generated, the list of buildings and the surface area of the village were returned, and a fitness was assigned based on these factors.

Historical fitness function

The first approach was to base the entire fitness function on historical villages, more specifically, medieval villages from around the 13th century. This function, and the villages generated using this fitness function will be referred to as "Historical". This period in history was chosen because the technology of that time seemed most equivalent to the simple tools used by someone starting a game in Minecraft. Basing the fitness function on historical data meant that the rates of the different buildings as compared to the population were based on research. During the 13th century, there was one acre of land necessary to sustain one person, and every third field was left fallow [35]. Therefore, one acre of farmland (or 14 fields) should be available per one population, and 7 pastures. The average family in these primitive villages would have had around five members: two parents, and three children [36]. Therefore, the fitness function was set to one house per 5 population. Roughly one in four working people was some form of craftsman [37], and since there were 2 working people per house, and five people per house, there would be one craftsbuilding per 10 population. The final factor that went into this fitness function, was the spread of the buildings. Medieval villages were built quite close together: 20 to 30 houses per acre. Therefore, the fitness functioned valued that amount of buildings per acre.

Game-based believable fitness function

While the historically based fitness function does have history to argue for its believability, there is reason to assume that this fitness function will not be perceived as believable when someone would be playing Minecraft. The reasons for this are that the number of fields necessary for one population would far outweigh any other landuse, and that players would assume every village to have more than just one or two types of crafters. Therefore, another fitness function was made: one that took the Minecraft game dynamics into account while basing fitness assessment on historical data. The most important thing that changed was the number of fields: instead of 14 fields per person (which is what one acre of farmland per person would come out to), it was changed to be one field per 5 population. This was done, because it can be reasoned with game-mechanics that one field is enough to sustain 5 people within Minecraft's systems.[1] The pastures were still considered as half the number of fields. The amount of population per house was changed from 5 to 3: normally, only one villager would sleep in a place with a bed, but with the houses being as big as they are, it seemed like multiple people per house would be more believable. Since there were fewer fields and pastures, it seemed like more crafter-type buildings could be placed. One type of crafter per 4 population seemed like a good number. Since it seemed like more than one fountain would be bad for believability, the fitness function was made to prefer only one fountain. Finally, the spread was reconsidered. Since it seemed like around 5 was around the maximum amount of structures that could be placed in one acre, this was set as the preferred spread for this fitness function. This fitness function and its related villages will be referred to as "Balanced".

Structures-only, spread-only, and random

In order to effectively test whether the fitness functions could improve believablity, other cases were explored. From the game-based believable fitness function, two aspects were split off to be explored on their own: the structures-based aspect, and the spread-based aspect. The structures-based aspect took all the calculations in the fitness function that had to do with the structures that were being build in relationship to the population amount, and only based the fitness on that aspect. The placement of these structures on the map was done randomly. This will be called "Need-based". The spread-only fitness function determined the type of building that would be placed next randomly, but tried to make it fit the spread-aspect of the game-based fitness function. This is referred to as "Spread-based". These two fitness functions were build to be able to compare the Balanced fitness function to, to see whether the complete fitness function would be better than the individual aspects of this fitness function. The final generator that was build was the random village-generator: it would place random buildings on random spots. (Referred to as "Random".) This was the most extreme comparison case.

[1]One square of farmland produces one wheat every two days. Three wheat are necessary for making one bread, and three bread are enough for filling the entire hunger bar (which will be considered to be the hunger one would get from not eating an entire day). Therefore, every villager needs 18 squares of field, to sustain them. A farm has on average 88 squares of field. $88/18 \approx 5$ people fed from one field

Chapter 5

Experiments and Results

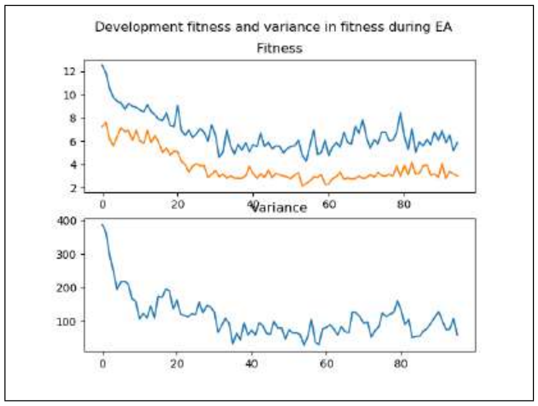
In this section, two experiments are described. First, the training of the evolutionary algorithm, and the parameters that were set are described in Section 5.1. The results of the experimentation with these parameters are discussed. After that, the results of the trained village generators are judged by humans on perceived believability through a survey. This is shown in Section 5.2. How the survey was set up, as well as the results of this survey is also discussed.

5.1 Training the EA

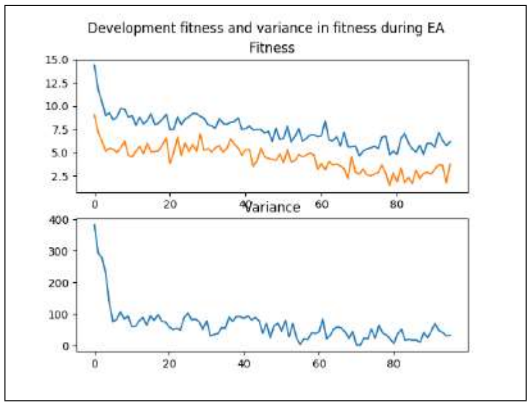
In this section, the way the evolutionary algorithm was trained is described. First, the parameters of the evolutionary algorithm are discussed, then the training process is described, and after that the results of the training will be shown and explained.

5.1.1 Parameters

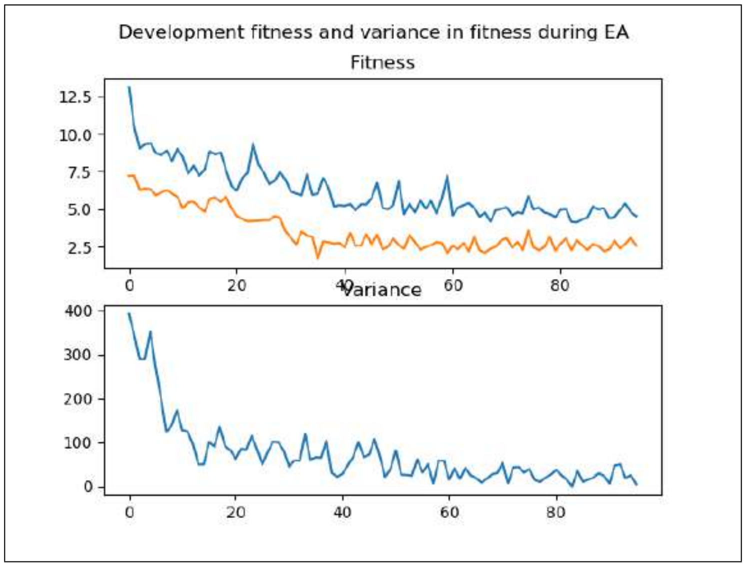
The evolutionary algorithm was set up with a population size of 15. This seemed to provide a good balance between enough variability, and training that was not too expensive in terms of clock-time. The number of generations was set to 100, since the fitness seemed to have leveled off at that point looking at the produced graphs. The outcome variable that is referred to as fitness through-out this thesis could maybe better be described as inverse fitness: a lower fitness is better in this thesis. Three different approaches for setting the value of the mutation rate were considered: a flat mutation rate of 50%, a flat mutation rate of 20%, and a mutation rate that started high at 50%, but decreased by 2% every 5 generations, ending in a mutation rate of 10% after all generations. This last approach to the mutation rate seemed to provide good results in literature [11]. All of these mutation rates were tested for the game-play based fitness function, since this was considered the most important and promising fitness function. The decreasing mutation rate made for the most stable mutation graph, so this mutation rate strategy was chosen for the further experiments. The graphs based on which this decision was made can be seen in figure 5.1. In this figure, as well as all following figures showing fitness graphs, the upper graphs show the fitness, with the best individual in orange and the average of the population in blue. The lower graph shows the variability. To reiterate: for this thesis, a lower fitness is better. Figure 5.1a shows a more irregular curve of the fitness, while the variance in figure 5.1b drops off very quickly. Figure 5.1c however, shows a more gradual decrease in the variability, and a gradual improvement of the fitness. Therefore, the decreasing mutation rate was used in all further training with the



(a) *Mutation rate of 50%*



(b) *Mutation rate of 20%*



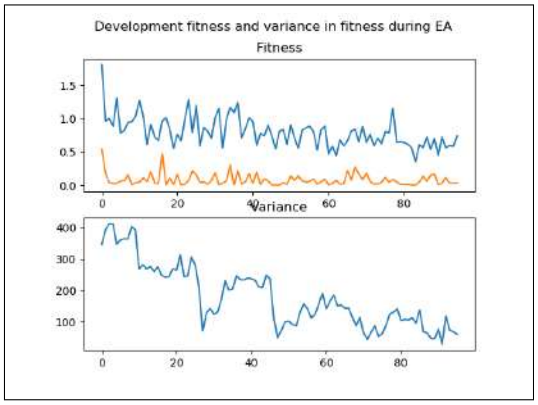
(c) *Mutation rate decreasing from 50% to 10% over the generations*

Figure 5.1: *Average fitness (blue), best fitness (orange) and variance for different mutation rates*

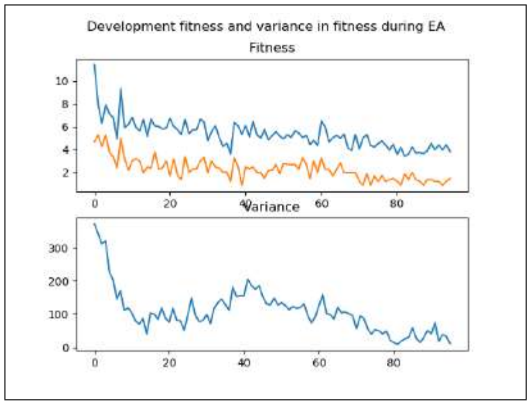
different fitness functions.

5.1.2 Training process

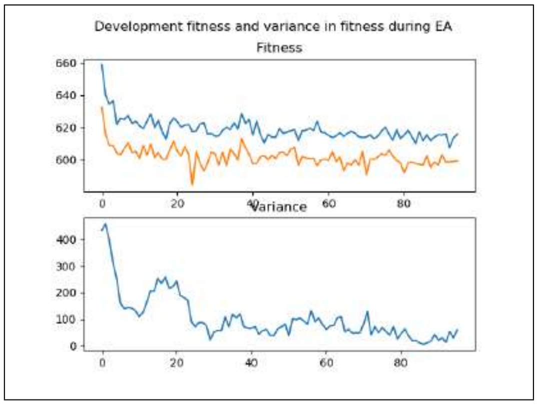
The evolution of the generators with the different fitness functions was done on a personal laptop. The laptop had a Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz processor, and 16 GB RAM. The duration of the training process differed for the different fitness functions: of the two aspects that were investigated, the spread and appropriate placement aspect was the most expensive, the need-simulation was less expensive. The Needs training took about 4 minutes per 5 generations, and the Spread, Historical and Balanced training took about 2 hours and 15 minutes per 5 generations.



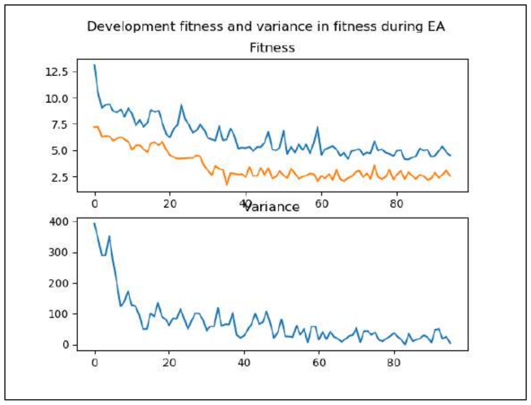
(a) *Fitness function only based on spread*



(b) *Fitness function only based on needs*



(c) *Historical fitness function*



(d) *Balanced fitness function*

Figure 5.2: *Average fitness (blue), best fitness (orange) and variance for different fitness functions*

5.1.3 Results

This section is split by types of results: first the quantitative results are discussed, including the shape of the fitness curves, then the qualitative results are shown, such as how the results of the different generators look.

Quantitative results

Figure 5.2 shows the graphs of the development of the variance and fitness using the different fitness functions. It can be deduced from these graphs that not all of these fitness functions were well-suited to this evolutionary algorithm set-up. Figure 5.2a, for example, shows a relatively stable fitness curve for the best individual, while the average lags behind. Spread is mostly determined by a single gene off the 31 gene long genome (the last gene, determining the search distance), so it is likely that through randomness the optimal value of this gene was reached quickly. Figure 5.2c also shows an interesting finding: the value fitness here is orders of magnitude larger than that of the other fitness functions. This is mainly because the demands made of the fitness function are not quite achievable for the generator: the amount of houses and fields and pastures, combined with the amount of structures asked for per square meter are not physically possible for this generator to achieve. Despite that, the fitness does seem to slowly improve over the generations.

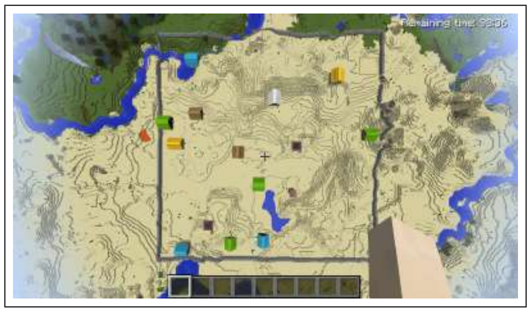
Qualitative results

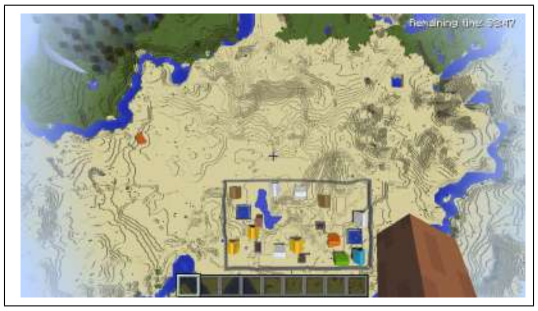
After training, the genes produced by the evolutionary algorithm were used to set the variables of the different generators, and villages were made using these generators. Some images of these villages in the dessert biome are shown here in bird's-eye view 5.3. For clarity of these images, the different house-like building have been given colored roofs. As was expected, both the Random and Needs generators occupied a larger area than the other generators, since they put buildings down in the entire box, instead of searching for appropriate places close to already build buildings (see figures 5.3a and 5.3c). Both Spread and Random had more different kinds of buildings, as can be seen from the variety of colored roofs. Since all buildings had the same chance of being selected, there were also more different buildings being built (see figures 5.3a and 5.3b). Historical seems to have a particular preference for pastures, which are the rectangular fenced in areas 5.3d. This is explicable by the fact that the fitness function of this village demands more pastures and farms than is possible, but it demands less pastures than fields, so it is easier for the generator to satisfy this criterium. The Balanced village does seem to combine the properties of the Spread- and Need-based villages: it has a similar variation in buildings as Need-based, but on an area and spread out in a similar way to Spread-based (figure 5.3e).

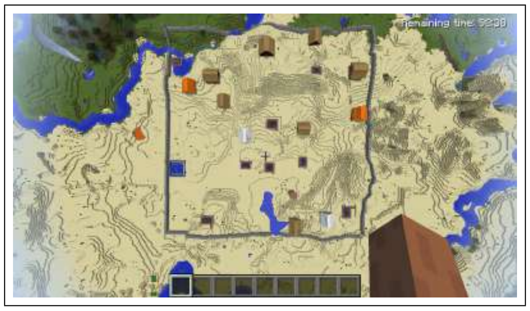
While the different villages look significantly different in bird's-eye view, these differences are not obvious when walking through the villages, as if playing Minecraft. Figure 5.4 shows images of how villages built in the same biome look like when exploring them on foot. While Historical (figure 5.4d) stands out with the amount of pastures, and Needs and Random (figures 5.4c and 5.4a) are more spread out than the other villages, the differences are subtle. This could pose a problem.

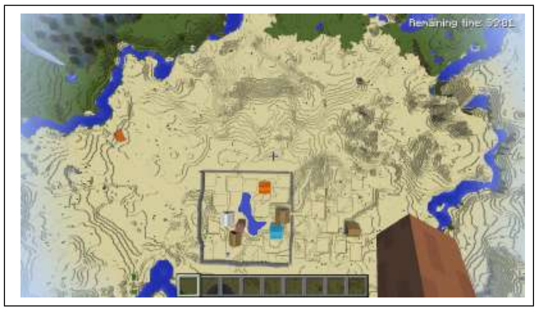
5.2 Validating Believability

In this section, the survey made to compare the villages resulting from the different fitness functions will be discussed. First, the setup of the survey itself will be explained. After that, the


(a) *Random village*


(b) *Spread-based village*


(c) *Needs-based village*


(d) *Historical village*

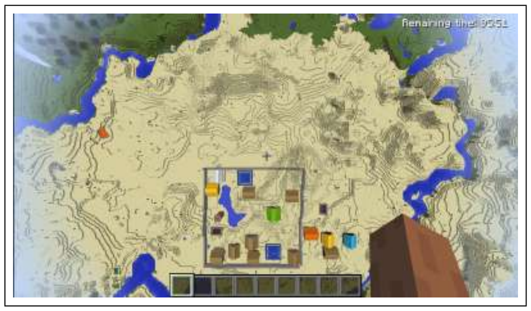
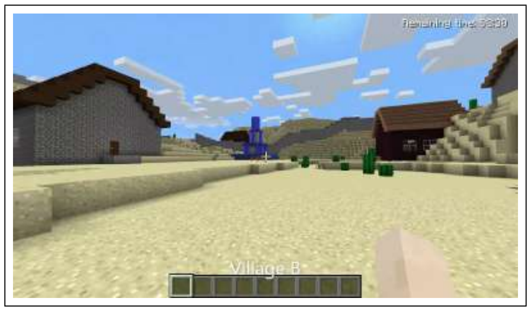
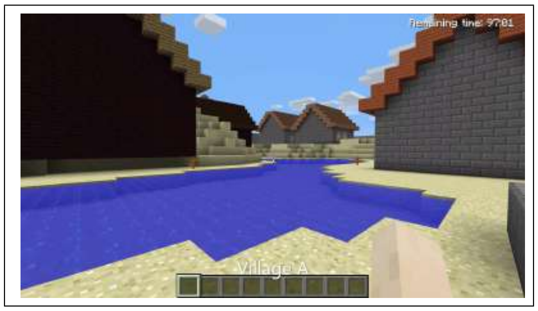

(e) *Balanced village*

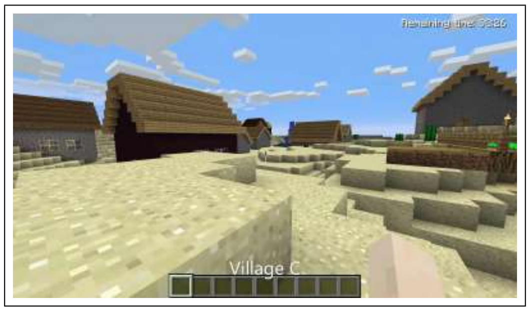
Figure 5.3: *Bird's-eye views of the villages resulting from the different fitness functions*



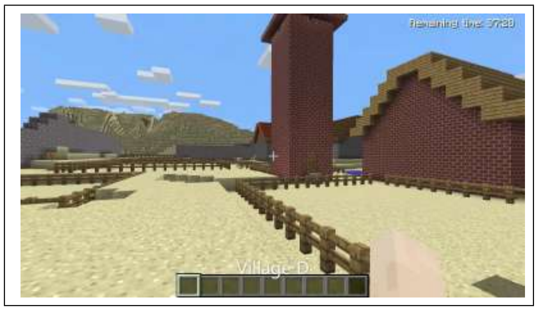
(a) *Random village*



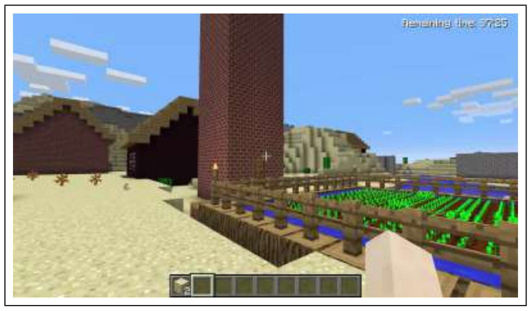
(b) *Spread-based village*



(c) *Needs-based village*



(d) *Historical village*



(e) *Balanced village*

Figure 5.4: *Player view of the villages resulting from the different fitness functions*

results of the survey, as well as a discussion of these results, will be shown.

5.2.1 Survey

On three different maps with different biomes, the different villages were generated by setting the variables to the genes of the best performing individual of the final generation with a specific fitness function. For each of these 15 villages, a short clip was made, using cheats to be flying in Minecraft to show an overview of the village. For each map, these 5 clips were combined into one YouTube video of about 3 minutes, that would show the 5 different villages that were the result of the 5 different fitness functions. The three biomes that were chosen were: a dessert, which is a very flat biome, a forest/mountainous biome with water, and the very uneven mesa biome, which has large plateaus with canyons between them. Impressions of these biomes can be seen in figure 5.5.

In the survey, every participant was first asked some demographic questions, as well as questions about their experience with games in general and Minecraft in particular. After that, they were shown the first video and asked to rank the five villages on the aspects of fun, immersion, and realism. These specific aspects were asked about because they are related to believability, but were considered to be easier for people to form an opinion about. Immersion and experienced realism are two important aspects of believablity, and perceived fun is what is expected to follow from believability, as well as an important aspect of creating content for games in general. The order in which the villages were shown was random, and the villages were just named "A" up to "E". At the end of the survey, there was a chance for the participants to leave comments on the survey. The survey was shared to DKE students, participants of GDMC through the discord, and to some personal friends of the researcher.

5.2.2 Results

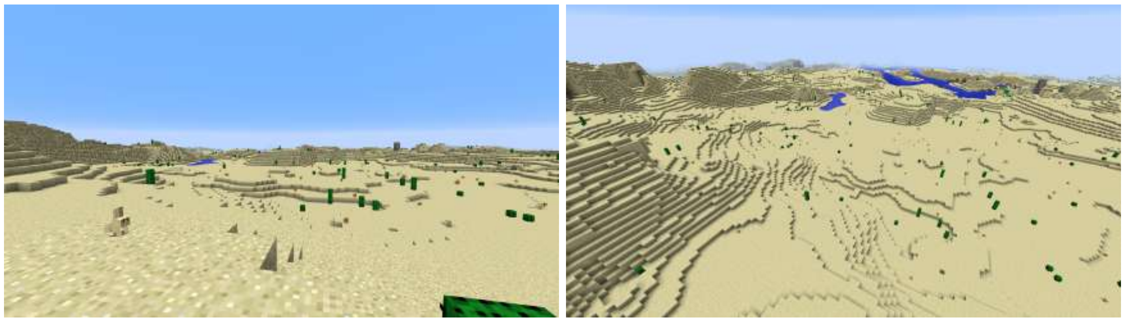
In total, 51 responses were received, of which 31 decided to complete the entire survey, not only the first village. Of these, 44 individuals were male, 5 were female, and the other 2 responses identified as non-binary or would prefer not to say. Most of the responses were by people in their late teens or twenties, with a few outliers on either side. More than half of participants played videogames at least multiple times a week, and more than three quarters played videogames at least multiple times a month. None of the participants had never played videogames. The overwhelming majority had some familiarity with the game Minecraft: 66.7% had played it for more than 100 hours, and 25.5% had played for between 10 and 100 hours.

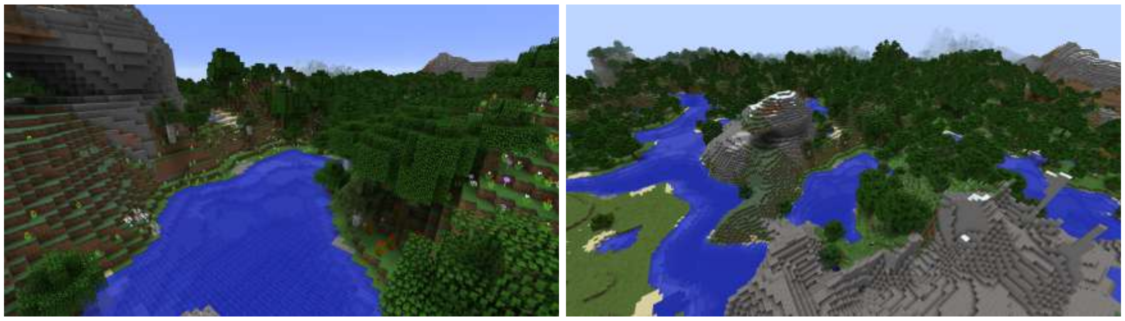
After the results were received, some data treatment was done. More specifically: the rankings for the three different questions were translated into scores, with a score of 5 on a questions for one of the five village types being the best, and 1 being the worst. These scores were added up per individual for all three questions, and averaged. The results of this are shown in Table 5.1. For all following analyses, the value of α was 0.05.

Comparing Questions

After the averages were calculated, the effects of the three different questions were investigated individualy. First, ANOVAs were done for all these questions, on all these maps. The p-values of these ANOVAs can be seen in Table 5.2.

As can be seen in Figure 5.2 all different questions were significant on the desert map, while only the realism question was significant in the other maps. The the difference between the Balanced and Random types of villages was the most relevant to this research, so a t-test was


(a) *Desert biome*


(b) *Forest/mountain biome*

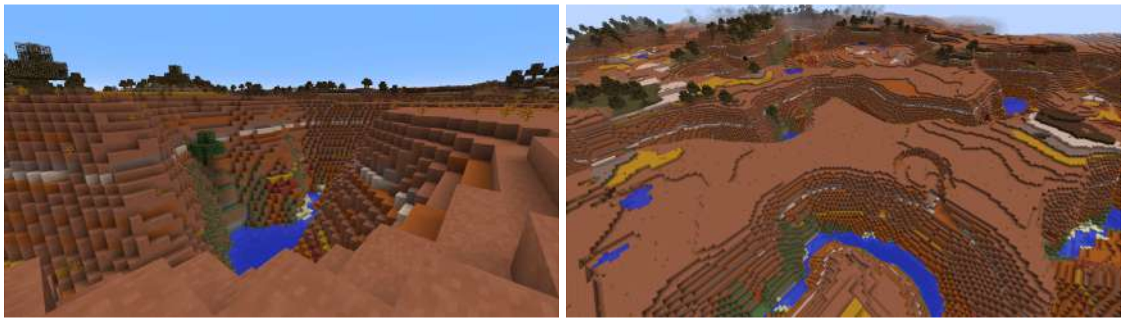

(c) *Mesa biome*

Figure 5.5: *Impressions of the three different biomes*

performed to see whether the difference between Balanced and Random was significant for any of the significant questions. The results are shown in Table 5.3. For all of these, the difference between Balanced and Random was significantly different. A Pearson correlation test was also run to compare values of the different questions to each other. This was only done on the desert village (village 1). The results can be seen in Table 5.4. All of these questions were significantly correlated for this map, which does indicate that fun, immersion and realism do seem to go together for this type of biome.

Total Score

As can be observed from Figure 5.1, the Balanced village type seemed to perform quite well in both village 1 and 2 (the desert and mountainous/forest areas). Balanced has the highest score in both of those villages. In village 3, Balanced did not perform as well: there it only has the second

Type	Village 1	Village 2	Village 3
Balanced Total	12.039	10.452	9.710
Fun	4.216	3.323	2.903
Realism	3.824	3.581	3.548
Immersion	4.000	3.548	3.258
Historical Total	6.138	8.194	8.161
Fun	1.804	2.742	2.806
Realism	2.176	2.839	2.581
Immersion	2.157	2.613	2.774
Spread Total	9.275	9.806	8.258
Fun	2.961	3.226	2.774
Realism	3.216	3.516	2.774
Immersion	3.098	3.065	2.710
Needs Total	9.941	8	10.710
Fun	3.431	2.807	3.548
Realism	3.235	2.548	3.613
Immersion	3.275	2.645	3.548
Random Total	7.608	8.548	8.161
Fun	2.588	2.903	2.968
Realism	2.549	2.516	2.454
Immersion	2.471	3.129	2.710

Table 5.1: *Average and Variance of Scores*

Map	Fun	Realism	Immersion	Total
Village 1 (desert)	1.44×10^{-20}	2.97×10^{-9}	6.28×10^{-12}	5.76×10^{-17}
Village 2 (forest/mountains)	0.391	0.00189	0.0540	0.0306
Village 3 (mesa)	0.188	0.000868	0.0579	0.0117

Table 5.2: *ANOVA p-values*

highest score, after Needs. Historical did not perform well: it is always either worst or second worst in terms of score. Spread seems to perform better in village 2, and Needs seems to perform better in village 3. This could indicate that different parts of the Balanced generator are more important in different biomes.

ANOVAs ANOVAs were performed to investigate whether there was statistically significant difference between the different groups. The results of these ANOVAs can be seen in Table 5.2. All ANOVAs indicated that there was a significant difference between the totals of the groups. The difference seems to be the most certain for village 1: the figure shows a p-value of 5.76×10^{-17} for this village, which is much lower than α. Both village 2 and 3 also showed statistical significance, but here the p-values were higher (0.03 and 0.01, respectively).

T-tests Because it was shown that there was significant difference within all maps, the question became which village types had the significant difference. In order to find that out, t-tests were done. The results of t-tests in village 1 can be seen in Table 5.5. All of the differences between the groups that were investigated were statistically significant. For village 1, it was decided to

Map	Type	P-value
Village 1 (desert)	Fun	3.05×10^{-11}
Village 1 (desert)	Realism	3.68×10^{-6}
Village 1 (desert)	Immersion	1.16×10^{-8}
Village 2 (forest/mountains)	Realism	0.00557
Village 3 (mesa)	Realism	0.00168

Table 5.3: *P-values of two-tailed t-test of the difference between Balanced and Random*

Measures	Pearson correlation coefficient	P-value
Fun and Realism	0.657	1.652×10^{-7}
Fun and Immersion	0.757	1.315×10^{-10}
Realism and Immersion	0.724	1.997×10^{-9}

Table 5.4: *Pearson correlation coeffient and p-values for the different measures in village 1*

focus on the performance of Balanced, since this seemed to be the most promising village type. It seems that Balanced is significantly better than all other village types for this particular map: the two-tailed p-value is lower than α in all comparisons done. The Historical village type also was significant, but in this case significantly worse than Random (which indicates that it is likely the worst performing village, since Random has the second lowest average score (see Table 5.1)).

Type	Type	P-Value
Balanced	Random	1.72×10^{-10}
Balanced	Spread	0.000259
Balanced	Needs	0.00138
Historical	Random	0.0137

Table 5.5: *T-tests for village 1*

For village 2, the averages seen in Table 5.1 seemed closer together, so it was less likely that there would be significant difference between the different village types. It was the case that Balanced was significantly better than Random, as can be seen in Table 5.6. Spread, which also had a higher average score than Random, was not significantly different however, and Balanced was not significantly different from Spread. Therefore, it is not possible to state that Balanced performed best in this case, only that it performed better than the Random generator.

Village 3, the mesa biome, did not show a significant difference between Balanced and Random (table 5.7). Needs however was significantly better than Random, which is interesting, since the Needs generator generated houses that would only depend on the needs of the city, not on spread. This seems to indicate that for the mesa biome, the need-based AI matters more than the placement AI. It is also possible that the close-together villages that the Balanced evaluation function prefers are not very well-suited for making believable villages in this particular biome.

Type	Type	P-Value
Balanced	Random	0.0473
Spread	Random	0.193
Balanced	Spread	0.446

Table 5.6: *T-tests for village 2*

Type	Type	P-Value
Balanced	Random	0.0824
Needs	Random	0.00583

Table 5.7: *T-tests for village 3*

Chapter 6

Conclusion and Discussion

To reiterate, the problem statement for this thesis is the following: "How to develop a procedural village generator that produces (more) believable villages for Minecraft-like games?" The research questions to this problem statement are:

- What existing ways of procedurally generating villages for videogames are there?
- How to build a simulation-based village generator for Minecraft-like games?
- How to optimize the parameters for this village generator for believability?

What existing ways of procedurally generating villages for videogames are there? In Chapter 2 answers were given to the first research question. The main subjects discussed in that chapter were those which were relevant for this research: search-based PCG, mixed initiative, and evaluating PCG content, as well as short descriptions of earlier work on subjects relevant to this thesis. Earlier research specifically for generating villages in Minecraft was also shown in Chapter 3, where multiple earlier submissions to GDMC were discussed.

How to build a simulation-based village generator for Minecraft-like games? The second subquestion was how the generator would be built. The generator was inspired by the needs-based AI from the Sims, with separate algorithms for determining the placement of new buildings. This seemed to provide satisfying results, but there were some parameters to tune. That is where the third subquestion came in.

How to optimize the parameters for this village generator for believability? The parameters were tuned through an evolutionary algorithm. On the results of this algorithm, a survey was performed. The results of the survey indicated that the tuned algorithm was in most cases better performing than random placement, and that in certain biomes this performed better than all other approaches. However, it also seemed to indicate that for other biomes, the factors that were selected for through the evolutionary algorithm might not achieve believability.

Some points of discussion could be brought up about this thesis. While it was argued that fun, immersion and realism are related to believability, it is possible that there are other important facets of believability that were not investigated. This would mean that the survey done does not completely reflect believability. Most of the problem comes from the fact that this concept is difficult to quantify, and is therefore a difficult quality to measure. However, the questions used did come close enough to measuring believability to be informative. Another problem with the design of the study, is that the differences in the different village types were not always clear for all participants, and that the participants sometimes had difficulty remembering which village was which. This was mentioned often in the comment box at the end of the survey, and in personal messages to the researcher. To mitigate at least the second one of these problems, screenshots of the villages could have been provided to the participants along with the video, so it would have been easier to remember which is which. Another limitation of this study is that there was only one village generated per type per biome. It is therefore possible that differences in survey scores for the different villages is due to the generator randomly performing better in one biome than the other, and not due to the generator being better suited to one biome over another. However, when the generator was observed generating villages multiple times, it did not seem to differ wildly in quality, therefore this is considered less of a concern.

Chapter 7

Further research

In this chapter, the possible future work that could be investigated based on this research is discussed. This future research falls into a few categories: changes to the need-based simulation, making the results of the generator more aesthetically pleasing, expanding the search, and application to different fields.

7.1 Simulation

Some experimentation could be done to expand the current simulation method. Firstly: an interesting avenue to explore could be simplifying certain aspects. Currently, with every tick the village's needs lower less depending on how many buildings are already satisfying a specific need. In the original algorithm, the amount by which needs were lowered each tick was not dependent on what a sim already had in terms of resources, but was only dependent on what action the sim was doing, and the general depletion over time [24]. It would be worthwhile to investigate whether the satisfaction aspect of the needs-based algorithm is necessary at all: it might not even be required. If this aspect was left out of the simulation, there would be less parameters to tune, which would be good if the same output quality was kept.

Another possible change to the algorithm, to make it more adaptive, is to change the needs of the village depending on the environment. A desert village, for example, could have a larger need for water, and a village built next to a place where a lot of wild animals spawn would have less need for food. A bigger difference would be made by changing what kinds of buildings satisfy different needs in different environments. Maybe the village close to a large body of water has mainly fishing buildings, because that is where that kind of village would naturally get the most food from, while the mountainous village has some form of rice fields, and the village in the plains has mostly pastures. By taking the environment more into account, the adaptability would improve, and therefore it can be assumed that the believability would improve as well.

The needs that were chosen for the current generator are quite arbitrary. There was some consideration on what every village would need (enough shelter and food for all its inhabitants, for example), but many of the other needs were chosen because they were achievable with the set of buildings that was used. The set needs could easily be expanded. Additions that would be advised to make are: the need for security (satisfied with watchtowers, a village wall), transportation (train tracks, or easily navigable paths), beauty (flowerbeds). Other buildings could also be added. The need "social", for example, only has one structure associated with it, which does not allow for much expressivity.

The costs of building a structure could be made more complex. Currently, the only factor to the costs is the steepness of the terrain on which a new structure will be built. However, by making the cost of building a structure relate to the type of building built (a pasture does not take too much resources, while a tower might take much more), the environment that is already there (it might cost more to build a field with water or a fountain in a dessert) and the other buildings surrounding it (a lumberjack might decrease the costs of nearby wood construction), the simulation could get more complex and realistic. This amount of realism could translate into better believability.

The generator currently only yields its results after all ticks have been completed. However, an advantage of this generator that can contribute to its believability is the ability to keep adding buildings to an already existing village, thereby giving the illusion of organic growth. While difficult to implement in MCEdit, it would be possible to give the results of the generator at different time-steps, which would mean that how the village is developing over time can be shown. This would be especially interesting if it could be achieved during gameplay, as it would make a village feel more alive if you were to leave it, and find that it had grown and developed when you would return to it.

7.2 Aesthetic Improvements

Improving the aesthetic quality of the generated villages would likely make them more appealing. The first thing that could be done in order to achieve this goal, is to make the buildings in the village more visually different. As the generator is now, all house-like buildings look very similar, and that can be perceived as boring. It would make for better looking villages if the generator was capable of making places like the crafters and the blacksmith look different from each other and from the house. It would also improve the generator to be able to add small aesthetic differences to the houses on other places than just the inside. Perhaps for certain houses different heights were chosen, or they have a differently shaped roof, or some buildings have a smaller footprint but are multiple stories. By making the aesthetics of the village more varied, a more believable village would be generated, since it does make sense that individual villagers would have different tastes. This could even be adapted to at which tick a particular building was built: earlier buildings could look more simple, while later buildings could be more elaborate (because a larger village could mean more wealth), or the other way around (people who have lived in the same place for longer have had more time to improve their buildings).

7.3 Expanding the Search

The search that was done on finding the right variables for the generator could be improved. It seemed that the Balanced fitness function was not well-suited for all biomes, since especially the search area variable did not perform as well in the mesa biome as in the desert biome. Therefore, it would be worthwhile to change the fitness function; either by using variables that work regardless of biome, or by adjusting the fitness function to the biome. Perhaps the most believable amount of houses per square mile is different depending on the biome, so that could be taken into account in the fitness function (biome dependent), or people prefer seeing all buildings on the same height level (regardless of biome). These changes to the fitness function could be explored.

The search that was done now was relatively limited in which variables of the generator it adjusted. It only changed the needs-based variables, and the search area. It would be interesting to investigate changing other variables such as those that influence the starting location for the village, or biasing the size of the buildings.

The research done in this thesis did not do an extensive optimization of the hyper-parameters of the evolutionary algorithm. It is possible that better results could be achieved more quickly, if these were set to different values. Therefore, it could be worthwhile to research the influence of different values for the hyper-parameters in further research.

7.4 Different Applications

The final category of interesting further research is testing out the generator on other applications. Other generative open world games might benefit from this type of village generation: a game such as Rimworld, where players build their own survivalist colonies from a top-down 2D perspective and can visit neighboring colonies could benefit from this type of village generator. Different buildings with different functions could be placed in a way that makes sense, and the earlier mentioned development of villages over time could enhance the experience in this game.

Another application that this village generator could have, is as starting off point for designing villages in non-generative games, or to design smaller villages in very large games. If the villages are used as a starting-off point, then generating villages can decrease the work-load on the designers, since they will only have to tweak the villages instead of building from the ground up. In the really large open worlds, having a couple villages generated in this way can provide a small interesting feature to travel towards, with very little development cost.

Bibliography

[1] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.

[2] J. Steuer, "Defining virtual reality: Dimensions determining telepresence," *Journal of communication*, vol. 42, no. 4, pp. 73–93, 1992.

[3] S. Parkin, "How designers engineer luck into video games," March 2019. [Online]. Available: https://nautil.us/issue/70/variables/how-designers-engineer-luck-into-video-games-rp

[4] Mojang, "Minecraft," https://www.minecraft.net/nl-nl, accessed: 19-08-2021.

[5] C. Salge, M. C. Green, R. Canaan, and J. Togelius, "Generative design in Minecraft (GDMC) settlement generation competition," in *Proceedings of the 13th International Conference on the Foundations of Digital Games*, 2018, pp. 1–10.

[6] J. Togelius, N. Shaker, and M. J. Nelson, "The search-based approach," in *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, N. Shaker, J. Togelius, and M. J. Nelson, Eds. Springer, 2016, pp. 17–30.

[7] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelbäck, and G. N. Yannakakis, "Multiobjective exploration of the starcraft map space," in *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*. IEEE, 2010, pp. 265–272.

[8] J. Togelius, M. Preuss, N. Beume, S. Wessing, J. Hagelbäck, G. N. Yannakakis, and C. Grappiolo, "Controllable procedural map generation via multiobjective evolution," *Genetic Programming and Evolvable Machines*, vol. 14, no. 2, pp. 245–277, 2013.

[9] J. Togelius, R. De Nardi, and S. M. Lucas, "Making racing fun through player modeling and track evolution," 2006.

[10] Y. Kaya, M. Uyar *et al.*, "A novel crossover operator for genetic algorithms: ring crossover," *arXiv preprint arXiv:1105.0355*, 2011.

[11] A. Hassanat, K. Almohammadi, E. Alkafaween, E. Abunawas, A. Hammouri, and V. Prasath, "Choosing mutation and crossover ratios for genetic algorithms—a review with a new dynamic approach," *Information*, vol. 10, no. 12, p. 390, 2019.

[12] J. Togelius, N. Shaker, and M. J. Nelson, "Mixed-initiative content creation," in *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, N. Shaker, J. Togelius, and M. J. Nelson, Eds. Springer, 2016, pp. 195–214.

[13] ——, “Evaluating content generators,” in *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, N. Shaker, J. Togelius, and M. J. Nelson, Eds. Springer, 2016, pp. 215–224.

[14] G. N. Yannakakis and H. P. Martínez, “Ratings are overrated!” *Frontiers in ICT*, vol. 2, p. 13, 2015.

[15] T. Archer, “Procedurally generating terrain,” in *44th annual midwest instruction and computing symposium, Duluth*, 2011, pp. 378–393.

[16] A. Cristea and F. Liarokapis, “Fractal nature-generating realistic terrains for games,” in *2015 7th International Conference on Games and Virtual Worlds for Serious Applications (VS-Games)*. IEEE, 2015, pp. 1–8.

[17] M. Frade, F. F. de Vega, and C. Cotta, “Automatic evolution of programs for procedural generation of terrains for video games,” *Soft Computing*, vol. 16, no. 11, pp. 1893–1914, 2012.

[18] J. Doran and I. Parberry, “Controlled procedural terrain generation using software agents,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 2, pp. 111–119, 2010.

[19] T. Lechner, P. Ren, B. Watson, C. Brozefski, and U. Wilenski, “Procedural modeling of urban land use,” in *ACM SIGGRAPH 2006 Research posters*, 2006, pp. 135–es.

[20] C. A. Vanegas, D. G. Aliaga, B. Benes, and P. A. Waddell, “Interactive design of urban spaces using geometrical and behavioral modeling,” *ACM transactions on graphics (TOG)*, vol. 28, no. 5, pp. 1–10, 2009.

[21] N. Kato, T. Okuno, A. Okano, H. Kanoh, and S. Nishihara, “An alife approach to modeling virtual cities,” in *SMC’98 Conference Proceedings. 1998 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No. 98CH36218)*, vol. 2. IEEE, 1998, pp. 1168–1173.

[22] T. B. Mizdal and C. T. Pozzer, “Procedural content generation of villages and road system on arbitrary terrains,” in *2018 17th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. IEEE, 2018, pp. 205–2056.

[23] M. B. Temuçin, İ. Kocabaş, and K. Oğuz, “Using cellular automata as a basis for procedural generation of organic cities,” *European Journal of Engineering and Technology Research*, vol. 5, no. 12, pp. 116–120, 2020.

[24] R. Zubek, “Needs-based AI,” *Game programming gems*, vol. 8, pp. 302–11, 2010.

[25] K. Dill, E. R. Pursel, P. Garrity, G. Fragomeni, and V. Quantico, “Design patterns for the configuration of utility-based AI,” in *Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC)*, no. 12146, 2012, pp. 1–12.

[26] D. Charypar, A. Horni, and K. W. Axhausen, “Need-based activity planning in an agent-based environment,” in *12th International Conference on Travel Behaviour Research (IATBR), Jaipur*, 2009.

[27] L. Nijland, T. Arentze, and H. Timmermans, “Representing and estimating interactions between activities in a need-based model of activity generation,” *Transportation*, vol. 40, no. 2, pp. 413–430, 2013.

[28] C. Salge, M. C. Green, R. Canaan, F. Skwarski, R. Fritsch, A. Brightmoore, S. Ye, C. Cao, and J. Togelius, "The AI settlement generation challenge in minecraft: First year report," *arXiv preprint arXiv:2103.14950*, 2021.

[29] C. Salge, C. Guckelsberger, M. C. Green, R. Canaan, and J. Togelius, "Generative design in Minecraft: chronicle challenge," *arXiv preprint arXiv:1905.05888*, 2019.

[30] "Generative Design in Minecraft (GDMC)," https://gendesignmc.wikidot.com/, accessed: 17-08-2021.

[31] M. Fridh and F. Sy, "Settlement generation in Minecraft," 2020.

[32] A. Brightmoore, "GDMC2020 ChronicleChallenge." [Online]. Available: https://github.com/abrightmoore/GDMC2020_ChronicleChallenge

[33] H. Jia, S. Ito, and R. Thawonmas, "ICE_JIT." [Online]. Available: https://www.dropbox.com/s/apd6utnngm6o3dl/ICE_JIT.pdf?dl=0

[34] "MCEdit." [Online]. Available: https://www.mcedit.net/

[35] A. Aposolides, S. Broadberry, B. Campbell, M. Overton, and B. van Leeuwen, "English agricultural output and labour productivity, 1250-1850: some preliminary estimates," 2008.

[36] P. Toubert, "The carolingian moment (eighth–tenth century)," *A History of the Family*, vol. 1, pp. 379–406, 1996.

[37] F. C. Lane, "Before the industrial revolution: European society and economy, 1000-1700," 1978.