Procedurally Generating Natural-Looking Villages in Minecraft with Ant Colony Optimization Algorithms

Master's Thesis

Tobias Deinböck

Supervisor:
池田心 (Ikeda Kokolo)

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)

August, 2023

Abstract

Artificial intelligence (A.I.) has been a actively studied area of research, especially in recent years. Among the large number of A.I. methods, procedural content generation (PCG) and agent-based simulations are two topics that have attracted much attention. The goal of PCG algorithms is to automatically generate game content (levels, maps, non-player characters, etc.) to save both time and expenses for human designers and artists. It is essential that a PCG method reliably generates content with the desired properties and qualities. To ensure this, PCG has become the subject of much academic work in recent years, and especially the game industry has been studying PCG methods a lot. Also at the center of much attention are agent-based simulations. Here, many small agents living in a bigger system follow certain rules, and by their individual actions and decisions, properties or structures are formed on the system level. One example of an agent-based simulation method are ant colony optimization (ACO) algorithms. These are optimization methods that are usually applied to find the shortest path through a graph. The ant agents mark the paths they find through the graph, and the shorter a path is, the stronger it is marked. Other ants are more likely to follow a path the more strongly it is marked, and mark it themselves when they then traverse it. As a result, after some time, the shortest path is the strongest marked one and all ants follow it.

Minecraft is a game whose worlds consisting of blocks are generated using PCG algorithms. In these worlds, there are villages, structures consisting of buildings and paths, that are also procedurally generated. However, we find that these villages do not look very natural. We define a village to be natural if it meets the following criteria: It has a clearly identifiable village center, houses are placed in locations that make sense depending on their function, paths adapt to the slope of the terrain, and paths follow a hierarchy. In our opinion, the default villages in Minecraft meet none of these criteria.

To generate more natural-looking villages, we propose using ACO algorithms. The idea behind it is that the life of the villagers is simulated by ant agents. Whenever a villager wants to go from one house in the village to another, it is the ants that find a path for the villager. Ants look for short and flat paths, which they mark with different strengths depending on a path's length and flatness. In the process, a network of ant trails is formed, which in turn is defined to be the path network of the village. However, in order to fulfill this goal, the classical ACO algorithms have to be modified, since they were not designed to find natural solutions, but optimal ones. We must redefine on which basis the ants make pathfinding decisions, and furthermore on which criteria the strength of the path marking depends.

To be more precise, we propose to proceed as follows. First, houses are placed stochastically, where the probability of a location depends on two factors, namely the flatness of the terrain and the proximity to the village center. Paths between houses are generated using ACO algorithms. Each of the villagers is simulated for by ants. These ants find a path through the village whenever the villager wants to move from one house to another. The shorter and flatter the path found by an ant, the more likely it is that other ants will follow this path later. Through repeated search for paths, a network of paths that connects all houses in the village by short and flat paths is formed. Placing houses and finding paths can be repeated to grow the village. When the growth of the village is complete, the houses and paths are placed in the Minecraft world. Houses near the center of the village are assigned different functions than houses farther out, and paths that have been traversed more often are built wider than paths with less traffic.

Villages generated by this method consist of houses that are closer together in the center of the village than outside, and a network of paths that connects all houses. We analyze how the length and flatness of the paths that the villagers (ants) choose through the village changes over time. The results show that the paths become shorter and flatter as time passes. This means that the network of paths formed by the villagers connects a house of the village to any other house in the village by a short and flat path, as desired.

Furthermore, people with different backgrounds are interviewed and asked to compare the default villages and the villages generated by this algorithm. The results confirm that the generated villages are more natural than the default villages, with especially the course and structure of the paths being perceived as more natural. Overall, 73.3% of participants find the paths generated by our method to be more natural, and the majority of participants with prior playing experience in Minecraft find the placement of houses and the general appearance of the village to be more natural.

Acknowledgments

First of all, I would like to thank my professor and supervisor, 池田心 (Ikeda Kokolo). He assisted me in finding the directions for my research, and he always helped me when I got stuck. From the beginning, he followed my project with great enthusiasm and showed genuine interest. This strongly motivated me to keep going and try my best. I will be eternally grateful for his support, sincerity, and thoughtfulness.

Likewise, I would like to thank my assistant professor, 薛筑軒 (Hsueh Chu-Hsuan), who was always there when I had questions regarding my research or just wanted to talk about random things. She helped me immensely and without her this thesis would not be what it is now. For this and for making everyday life in the laboratory more fun, I am very grateful.

I want to thank my family and friends for supporting this journey of mine from the very beginning and for always being there for me. In particular, I would like to thank my mother Christine, my father Robert, my brothers Alexander and Maximilian, and my friends Alexander, André, Andreas, Benjamin, Dominik, Markus, and Valentin. Without all of them I would not be the person I am today.

Lastly, I would like to express my special thanks to 王思旻 (Wang Ssu-Min) for accompanying me throughout my life in Japan and always standing by my side. She made every day special and is one of the reason I will never forget my time in Japan. From the bottom of my heart, thank you.

Contents

List of Figures

List of Algorithms

1. Introduction

Over the course of the last few years, artificial intelligence (A.I.) has become increasingly important. A well-designed A.I. can support people in many areas of life or even completely take some parts of their work off them. One particularly interesting type of A.I. is the principle of procedural content generation, a method of algorithmically creating game content. Especially in video games, this is used a lot because it makes it possible to obtain a large number of objects or other types of content in a short time. As video games become larger and more complicated, the need for such methods also becomes greater since it has become unfeasible for humans to create every part of a game themselves. The technologies used in video games keep advancing, and thus the quality and efficiency of these procedural content generation methods must keep progressing as well. As a result, this area has received more and more attention in academic research over the past few years. Researchers are trying to develop new and better algorithms to assist the video game industry and many other fields beyond that without sacrificing the quality of the output.

Another type of A.I. that is gaining importance are so-called agent-based simulations. The idea behind them is that many small individuals (agents) exist together in a larger system and perform actions independently, with each agent pursuing a specific goal. This leads to the formation of certain structures or properties on the system level without the agents actively trying to do so. An example are ant colony optimization algorithms, where many small ants move between two points. Each ant has its own decision-making ability but is influenced by the routes of previous ants and tends to follow them. The shorter such a route is, the more it influences other ants. Although it is not the goal of the ants, they over time form an ant trail that is traversed by almost all ants. In other words, a structure in the larger system is formed by many small individuals.

In this thesis, we will combine the principles of procedural content generation and agent-based simulations. We want many small agents to generate some form of game content through the actions and decisions they make. To be precise, we will use ant colony optimization algorithms to generate video game content. The game we want to create content for is Minecraft, and the content to be created is villages. Minecraft is a popular game whose worlds are made of simple blocks. In these worlds, there are villages which are structures made of buildings and paths. Like the rest of the entire Minecraft world, these villages are procedurally generated. However, we find that the villages do not look very natural, that is, they have too little resemblance to villages in real life. Specifically, we find that the placement of the houses within the village seems random, that the streets do not respect the slope of the terrain, and that the streets do not follow any hierarchy. Therefore, our goal is to generate more natural-looking villages than Minecraft's default villages.

While the principles of the algorithm we will present in this thesis can be applied everywhere, where information about the terrain is available and where it is possible to interact with the world, we deliberately chose to use Minecraft for three reasons. First, the structure of Minecraft's worlds is very simple, as everything is divided into equal-sized blocks. Second, there are well-developed tools like Amulet that allow us to manipulate the worlds using computer scripts. Third, there are already villages in Minecraft that serve as a good basis for comparison with the villages we generate.

To achieve this goal, the algorithm performs the following steps. First, all the necessary information is extracted from the Minecraft world. This includes, for example, the elevation of the terrain and the positions of water. Then, houses are placed near a predefined village center and where the terrain is flat. Next, the movement of villagers living in the houses is simulated using ant colony optimization algorithms. If a villager finds a path between two houses, other villagers are more likely to follow it if this path is short and flat. Similar to ant trails, this creates paths that connect all the buildings in the village. Placing houses and simulating villagers can be repeated to make the village grow naturally. New houses are more likely to be placed closer to the village center which is defined as the average position of all houses. Finally, the village is built, that is, houses and paths are placed in the Minecraft world. The type of house depends on its position within the village, for example, a church is placed in the center but a farm is placed farther outside. The paths where many villagers walked are also placed in the world. The more often a path was used, the wider it is built.

There is already several academic work on generating villages or cities, both in Minecraft and outside of the game. Some of these papers follow similar ideas and principles as we do, though none of their approaches include all at the same time. Furthermore, we are not aware of any work that uses ant colony optimization algorithms to generate villages or cities.

This thesis is structured as follows. In Chapter 2, the game Minecraft is introduced with particular focus on default villages. In addition, we carefully define what our understanding of a natural village is. Chapter 3 explains terms and examines academic works related to our research, noting differences between these papers and the methods of this thesis. Chapter 4 includes the methodology of this research. It explains in detail how the designed algorithm works and what the motivations behind the design decisions were. The results of the algorithm, that is, the generated villages, are evaluated and discussed in Chapter 5, which includes an analysis of a survey with 45 participants. Chapter 6 concludes this thesis.

2. Minecraft - A popular game with imperfections

As the title of this thesis suggests, we will generate villages in a video game called *Minecraft*[1]. In order to understand the inner workings of the algorithm later, this chapter introduces the game itself as well as terms closely related to the game. We will mainly talk about the villages generated in the game and explain their function, but also mention some points we do not like about them.

2.1. Development history and impact

The development of Minecraft began in 2009, as a side project of Swedish video game developer *Markus "Notch" Persson*, which he worked on in his spare time. As he says himself, Minecraft was heavily inspired by games such as *Dwarf Fortress*[2], *RollerCoaster Tycoon*[3], *Dungeon Keeper*[4], and most notably *Infiniminer*[5] [12].

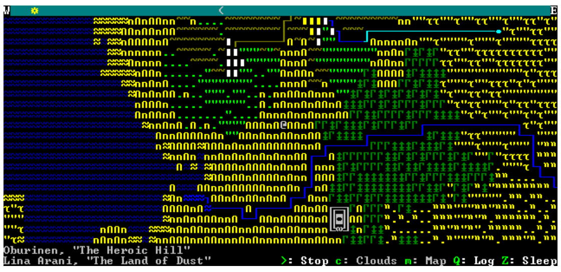
Dwarf Fortress is a single-player game set in a fantasy world. The player controls either a dwarven outpost or an adventurer in a randomly generated world characterized by individual civilizations, each with their own history spanning decades, hundreds of cities, caves, and various wildlife. While the game has no specific goal, the player attempts to build a strong and prosperous fortress as a dwarven colony, and explores and interacts freely with the world as an adventurer [2]. What makes Dwarf Fortress stand out from the other games mentioned is its graphics. The game is two-dimensional and the world, viewed from a bird's eye view, as well as everything contained in it, consists of simple, colored text icons (see Figure 2.1(a)). Even though these graphics make the game look simple at first glance, under the surface it is a very complex and well-designed game. This is reflected in its success: When the game was made available on Steam in early December 2022, it sold almost 500,000 copies within a month [38].

RollerCoaster Tycoon is a series of management simulation and sandbox games, the first of which was developed by Chris Sawyer in 1999 (for sandbox games see 2.2). The goal of the games is to build a successful amusement park with attractions such as roller coasters and stores (see Figure 2.1(b)) [14].

Dungeon Keeper is a series of real-time strategy games first released between 1997 and 1999. In the games, the player controls an anonymous entity known as the Keeper and builds an underground complex of rooms, doors and traps, and uses fiendish creatures to defeat invading heroes (see Figure 2.1(c)) [9].

Infiniminer was developed in 2009 by Zachary Barth in his spare time. With his game he invented a new way to represent the game world, using block-shaped entities from which the world is procedurally generated (see Figure 2.1(d), for procedural generation see Section 3.1). Players can mine and place these blocks, thus creating arbitrary constructions insofar as the world's block structure allows [26].

[1]Official website: `https://www.minecraft.net/`.
[2]Official website: `https://www.bay12games.com/dwarves/`.
[3]Official website: `https://www.rollercoastertycoon.com/`.
[4]GOG: `https://www.gog.com/en/game/dungeon_keeper/`.
[5]Official website: `https://www.zachtronics.com/infiniminer/`.


Oburinen, "The Heroic Hill"
Lina Arani, "The Land of Dust"
>: Stop c: Clouds m: Map Q: Log Z: Sleep

Source: `https://www.bay12games.com/dwarves/screens/adv44.png`.

(a) A typical scene from the game Dwarf Fortress. Pictured is the part of the world where the player (the gray @-symbol) is currently located. The beginning of a city can be seen in the north.

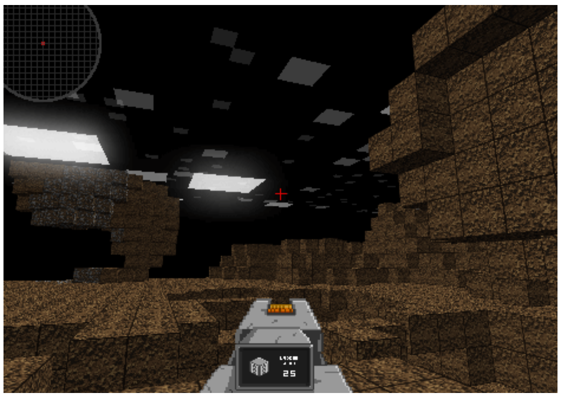
Source: `https://www.nme.com/wp-content/uploads/2020/09/RollerCoaster-Tycoon-3.jpg`.

(b) A view over a park in RollerCoaster Tycoon 3. This part of the game series was the latest in 2009, when Persson developed Minecraft.

84750
21406
Reinforced Wall. Hinders enemy tunnelling and improves your Dungeon. Health 100%

Source: `https://static.wikia.nocookie.net/dungeonkeeper/images/b/b0/Dungeon_Keeper_2_gameplay_screenshot.jpg/revision/latest?cb=20221106054937`.

(c) A constructed dungeon in Dungeon Keeper 2, the most recent game in the series in 2009.



Source: `https://www.zachtronics.com/infiniminer/infiniminer.png`.

(d) The automatically generated worlds of Infiniminer consist of blocks that the player can mine and place.

Figure 2.1.: Four of the games that served as inspiration for Persson when he developed Minecraft.

All of these games had features, characteristics or mechanics that Persson said he wanted to incorporate into his own project. From Dwarf Fortress, he wanted to bring the sense of depth and life that that game conveys so well to his own game. Beyond that, he wanted to make it even more interactive than Dwarf Fortress, to make surviving in the world even more exciting and immersive. In RollerCoaster Tycoon, he liked that he could quickly and easily build original and impressive roller coasters. Persson often spent hours daydreaming about complicated roller coasters, and he wanted convey that feeling in his project. Dungeon Keeper mainly influenced the atmosphere. Persson liked the common fantasy setting with caves lit by torches, and in his opinion, there were few games that captured the thrill of exploring a dark, creepy cave as well as Dungeon Keeper.

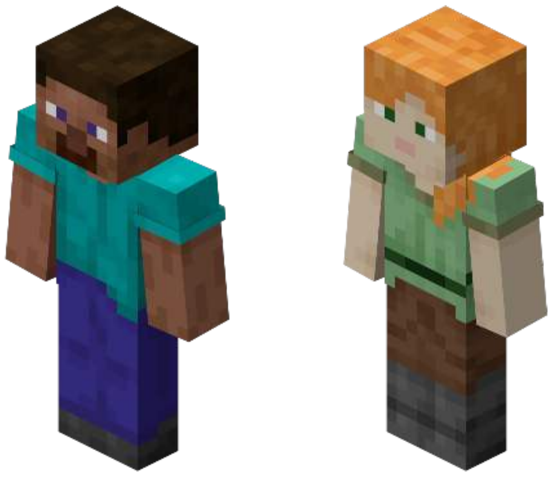
Those who are well-versed in Minecraft can clearly see Infiniminer's influence on Persson's project from Figure 2.1(d). A game set in world made of blocks that the player can freely interact with fascinated Persson so much that he immediately started programming once he was familiar enough with Infiniminer. He combined the fantasy setting of Dwarf Fortress, the easy-to-build ability of RollerCoaster Tycoon, the feel of dark, creepy caves from Dungeon Keeper, and the blocky world from Infiniminer, and released a prototype of a game he himself initially called an "Infiniminer clone" in early May 2009. After discussing it with some friends on a forum, he decided on a name for his game shortly after: *Minecraft* [12].

At the time, Persson could not have imagined the impact of his project. By May 2019 it had sold more than 180 million copies across all platforms, making it one of the most successful video game of all time [28]. According to the most recent official sales figures, as of April 2021, Minecraft has sold more than 238 million copies [37]. The immense popularity of Minecraft makes it a game that has many applications beyond just playing it. For example, there are publicly available tools to manipulate game data (see Section 2.4), contests are held related to the game (see Section 2.3.2), and even research is done on the game (see Section 3.5).

2.2. Game mechanics

We now know something about the development history of Minecraft and that it is the most successful game of all time. However, the answer to the most important question is still missing: What kind of game is Minecraft, and what makes it so special?

Minecraft falls into the category of a *sandbox game* [25]. In the video game industry, a game is called a sandbox game if it contains gameplay mechanics that encourage free play. These include, for example, an open world in which the player can move and act as they please, nonlinear storytelling, and NPCs (non-player characters) that are not or only slightly scripted. The term is a metaphor for a child playing in a sandbox, creating a world out of sand. A sandbox game thus contrasts with a game in which content is presented in its predetermined form and in a fixed order [4].



Source: `https://static.wikia.nocookie.net/minecraft_gamepedia/images/8/8f/Steve_%28classic%29_JE6.png/revision/latest?cb=20221026174119`.

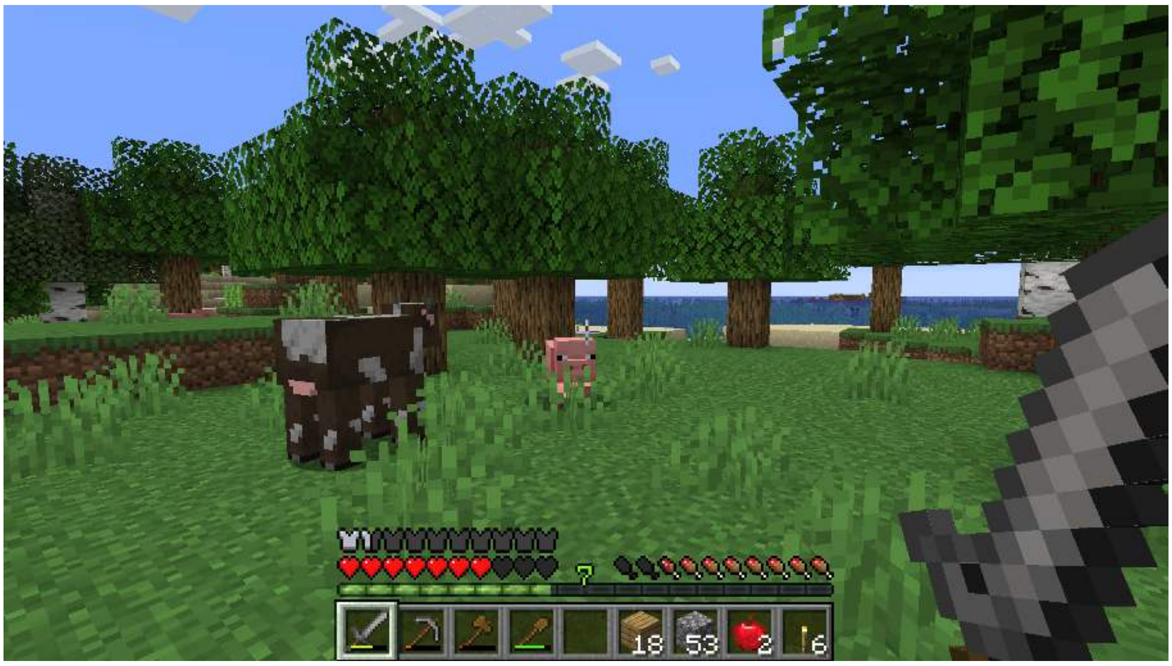
(a) Steve.

Source: `https://static.wikia.nocookie.net/minecraft_gamepedia/images/0/05/Alex_%28slim%29_JE3.png/revision/latest?cb=20221026174729`.

(b) Alex.

Figure 2.2.: Two of the players available by default. Apart from their appearance, they differ only in that the female player has thinner arms. There is no difference in gameplay. The player's appearance, which is called "skin", can be changed, either to official ones or to one of many available online.

In Minecraft, the player controls a character, which from now on will also be called the player (see Figure 2.2). At the beginning, the player is placed in a randomly generated world with an empty inventory [25]. Minecraft also has a multiplayer mode, which means that several players can join the same world and play together in it. They have a health bar of ten hearts and can receive damage from a wide variety of sources, such as falls, fire, mobs (see below), or other players when playing in multiplayer. Damage can be countered by armor and a full hunger bar, among other things. The hunger bar decreases over time, faster when the player runs, jumps, or swims, and can be refilled by eating food. There are four different difficulty modes, Peaceful, Easy, Normal, and Hard, and depending on the mode, the player can also die from starvation. In Peaceful mode, the health bar regenerates on its own and the hunger bar does not decrease [25]. A typical scene from a Minecraft world can be seen in Figure 2.3. If the player leaves the world, their inventory and the entire world are saved and they can continue playing where they left off the next time.



Source: Author's image.

Figure 2.3.: A typical scene in Minecraft. The player, who has already collected some weapons, tools, blocks, and food, is exploring a forest near the ocean. The health bar (red hearts) can be seen on the bottom left, and the hunger bar (drumsticks) is to its right. Above the health bar is the armor bar (chest plates), which shows how much armor the player is wearing, and below it is the experience bar, which shows the player's current experience level and progress to the next level (see below). At the bottom is the inventory's hotbar.

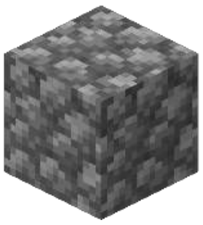
The world of Minecraft is a three-dimensional grid of cubes with a volume of one cubic meter, limited only vertically, and otherwise infinite in size. Each of these cubes is filled with a particular type of block, which are usually, but not always, perfectly cube-shaped themselves. At the time of writing this thesis, there are 879 different types of blocks. For example, there are naturally occurring blocks, such as grass, dirt, stone, wood, and ore, and blocks that the player must create, such as beehives, jukeboxes, and beacons (see Figure 2.4). The player can get resources from these blocks by mining them [25]. Often this resource is simply the block itself, but sometimes the player gets another type of block. For example, if they mine a grass block, they get dirt. Other blocks do not give the player blocks, but instead items that cannot be placed, such as a melon that breaks into melon slices. There are also blocks that can be mined but do not drop[6] anything (such as mob spawners), and blocks that cannot be mined at all, such as air or bedrock, the latter of which limits the

[6]In video games, the term "drop" is used to refer to items similar objects that some entity releases. It can also be used as a verb.

world below. Sometimes what a block drops depends on what the player uses to mine it. For example, if they mine stone with a pickaxe, the block drops cobblestone, but without this tool nothing drops. If the pickaxe is also enchanted with "Silk Touch", the block drops itself, i.e., as stone (for enchantments see below).

Source: `https://static.wikia.nocookie.net/minecraft_gamepedia/images/c/c7/Grass_Block.png/revision/latest?cb=20230226144250`.

(a) Grass block.


Source: `https://static.wikia.nocookie.net/minecraft_gamepedia/images/6/67/Cobblestone.png/revision/latest?cb=20220112085635`.

(b) Cobblestone.

Source: `https://static.wikia.nocookie.net/minecraft_gamepedia/images/2/29/Diamond_Ore_JE5_BE5.png/revision/latest?cb=20210326000237`.

(c) Diamond ore.

Source: `https://static.wikia.nocookie.net/minecraft_gamepedia/images/b/b7/Crafting_Table_JE4_BE3.png/revision/latest?cb=20191229083528`.

(d) Crafting table.

Figure 2.4.: Four of the most symbolic blocks in Minecraft.

The player has the ability to create new items (or blocks) by using existing items (and blocks) that they keep in their inventory. This process is called *crafting* [25]. This and the concept of mining blocks is the origin of the game's name [12]. For this purpose, the player can either use a 2-by-2 grid in their inventory or the 3-by-3 large grid provided by a crafting table (see Figures A.1(a) and A.1(b) in the appendix). By using a furnace and fuel such as wood or coal, the player can also smelt items and blocks into new items and blocks, for example making iron ingots from raw iron, which can then be further crafted into tools, armor, and more (see Figure A.1(c) in the appendix).

Different types of mobs (short for "mobiles") live in the world of Minecraft. There are hostile mobs that attack the player, neutral mobs that attack only when provoked, and passive mobs that never attack. Hostile mobs include zombies, skeletons, and creepers (see Figure 2.5), and examples of neutral mobs are bees and wolves. Passive mobs include most animals, such as pigs, sheep, cows, and chickens, and, among others, villagers [25], which are an important part of this thesis. In addition, there are also two boss mobs, which are many times stronger than normal mobs and unlike them cannot spawn[7] randomly, but must be deliberately encountered. They are many times stronger than normal mobs, but yield better rewards for the player.

Source: `https://static.wikia.nocookie.net/minecraft_gamepedia/images/5/5b/Creeper_JE2_BE1.png/revision/latest?cb=20191229172043`.

Figure 2.5.: Minecraft's most recognizable mob, the creeper.

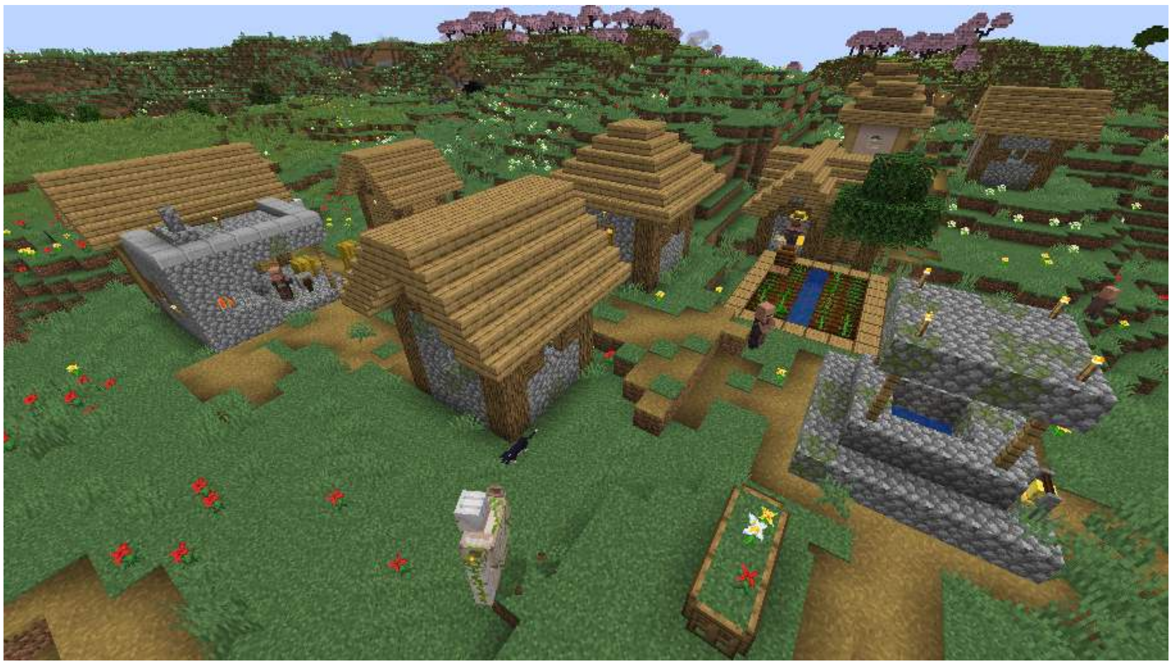
2.3. Villages

There are many so-called generated structures in the world of Minecraft. These are commonly occurring structures that appear in different biomes of the dimensions. Most of these structures appear to be man-made and many serve as sources of valuable and rare loot, making them extremely important for progressing through the game [16]. There are small structures like igloos and swamp huts, and there are large structures like mineshafts or pyramids.

Overworld (see Appendix A) structures include *villages* (see Figure 2.6). A village is a group or complex of buildings and other

[7] "Spawning refers to the creation and placement of players and mobs in the Minecraft world." [25]

smaller structures that is naturally generated. A village is inhabited by a variety of entities, each of which has a function. There are villagers (the people living in the village), cats as pets, iron golems for defense against hostile mobs, passive farm animal mobs, and wandering traders with their llamas. Sometimes a village is abandoned, in which case it is inhabited exclusively by zombie villagers. The player can get valuable loot in a village in several ways, such as looting chests or trading with villagers (see below) [25]. Since this thesis deals with villages, we would like to take a closer look at them below.

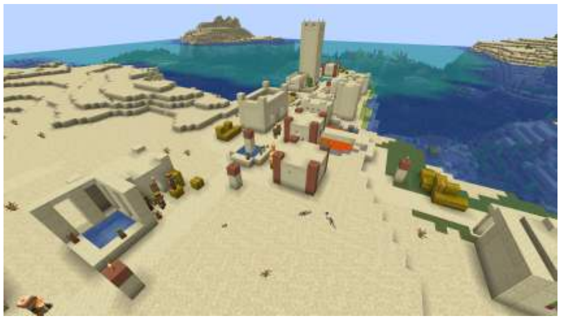


Source: Author's image.

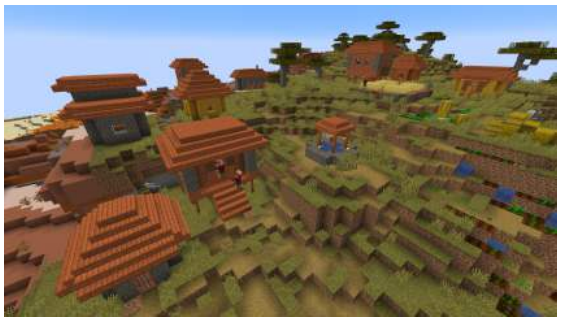
Figure 2.6.: A section of a typical village in a plains biome. Six normal houses, a blacksmith (left), a well (bottom right), a farm (left behind the well), and a fletcher house (house with the plus-shaped roof behind the farm) can be seen. Also visible are four villagers, an iron golem (bottom center), and a cat (behind the iron golem). On the horizon to the right of the big cherry blossom forest, a faint house of another village can be seen.

Villages are generated in biomes of the overworld, including the plains, savanna, taiga, snowy plains, and desert biomes. The biome in which the center of the village is located determines the appearance of all buildings and structures within the village. There are a total of five types of villages (see Figures 2.6 and 2.7). The number of villagers spawned when the village is generated depends on the number of beds in the village. Villagers spawn only in houses with beds, which does not include buildings such as the blacksmith or fletcher house shown in Figure 2.6.

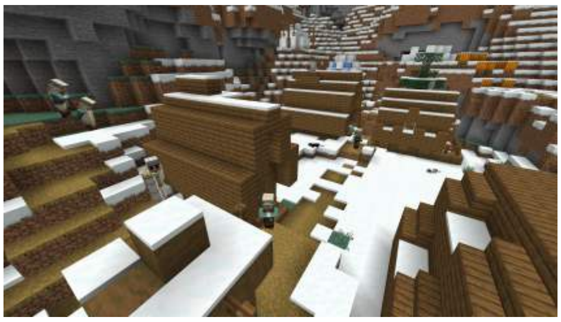
The number of buildings varies from village to village, and not every type of building has to be in a single village. While we think this is a very rough method, each building is chosen randomly from a pool of possible buildings, and no building has a higher probability than another. The number of decorative elements such as lamp posts, hay bales, or farms is unlimited, as these are generated only when no more houses can be placed. Perfectly straight paths that exclusively turn at right angles are generated between buildings and extend outside the village. They are placed at the same level of already existing terrain, but do not go below sea level. They replace only grass blocks, water, lava, sand, sandstone and red sandstone, and ignore all other blocks. Villagers and iron golems prefer to walk on


Source: Author's image.

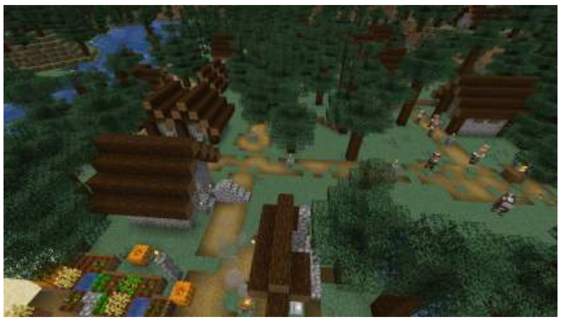
(a) Desert village.


Source: Author's image.

(b) Savanna village.


Source: Author's image.

(c) Snowy village.


Source: Author's image.

(d) Taiga village.

Figure 2.7.: These are the four other village types besides the plains village of Figure 2.6.

paths.

Every villager (except for the so-called nitwit and baby villagers) can have a profession, and depending on it they wear different clothes. A villager can hold a profession by claiming a job site block [25]. These blocks are placed when the village is generated and do not change their position unless the player mines them and places them somewhere else. Most of the job site blocks are generated in houses with special functions, such as the blacksmith or the fletcher house from Figure 2.6. The job site block of a farmer is generated next to farms, which means that there is no separate house especially for farmers. A job site block can only be claimed if it is within a village with at least one bed and if no other villager is already claiming it. An adult villager with no profession actively searches for unclaimed job site blocks. Each day, an employed villager (that is, a villager with a profession) follows their routine and works on their job site block according to the schedule's rules. Some do additional actions, such as farmers planting and harvesting crops and librarians examining bookshelves.

What makes villages so essential for progressing through the game is trading, a mechanic that allows the player to buy and sell items from and to villagers. Trading allows the player to obtain items that are otherwise difficult or impossible to acquire, such as diamond tools, diamond armor, and enchanted books [25]. This mechanic is further explained in Appendix A.

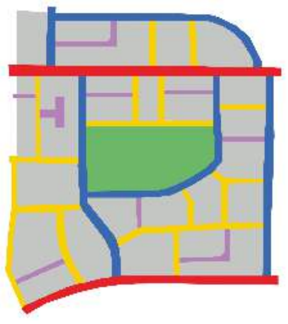
There are other mechanics in villages, such as villager breeding, raids, or zombie sieges [25], but they are not important for this thesis and therefore we will not discuss them further. Besides all these functional aspects and mechanics, villages are also important for simply adding to the aesthetics of the world and the fun of the game. Villages make the world feel much more alive and give the player a change from the otherwise almost all-natural world. It is fun to explore the villages and their houses, loot the chests and farms, protect the villagers from raids and zombies, and of course trade with the villagers.

All in all, we can see that villages are an important part of Minecraft.

2.3.1. Villages in Minecraft are not natural-looking

We have said a lot of positive things about villages in Minecraft. However, there are also some negative aspects about them. We could discuss a lot about whether the mechanics of villages are balanced or have other weaknesses, but we are specifically interested in something else. To be precise, we think that the villages in Minecraft do not look very natural. But before we can explain why we think this way, we need to define what defines a natural-looking village, and by that we mean one in real life.

There is a center in every village. This center is sometimes more and sometimes less complex and has different characteristics. Regardless, the center plays an important role in the life of the village. A large part of the tertiary industry is concentrated in the village center [31]. Tertiary industry is the sector of an economy, which includes services such as medical care, education, financial services, and transportation [19, 22]. Also part of the village center are larger residential complexes. In contrast, primary sector industries (such as agriculture, forestry, fishing, and mining) and secondary sector industries (such as manufacturing and construction) tend to be found outside the center [22, 31]. Moreover, the population density decreases rapidly the further away from the center one gets. To be more precise, there are usually more houses close together near the center of the village, while they are much more sparsely distributed further out [17].



Source: Adapted from `https://www.bristol.gov.uk/images/Subsites/Transport%20Development%20Management/Street%20Network%20Diagram.jpg`.

Figure 2.8.: An exemplary layout that follows road hierarchy. The order from major road to minor road is red, blue, yellow, purple.

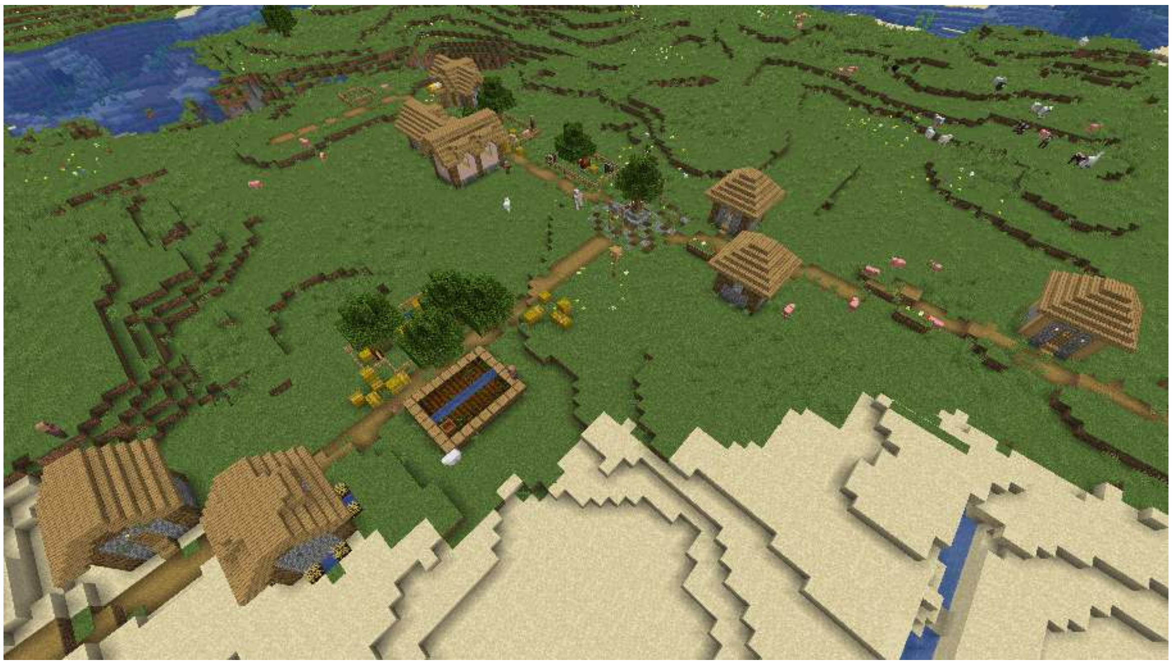
Another principle that villages follow is that of a so-called road hierarchy (see Figure 2.8). This is a form of classification of roads in which each type of road has a ranked position relative to all other types. Although there are several variants for the terminology, the basic principle is always a spectrum from major roads to minor roads. Major roads are often associated with strategic routes, heavier traffic flow, higher speed, and limited access to minor roads. Minor roads, on the other hand, are associated with lighter traffic flow, local routes, lower speed, more turnoffs, and more access to private homes [24].

These two concepts are important not only for planning and analyzing villages and cities in the real world, but also for creating believable, realistic, and natural-looking villages in digital media, such as video games. Many games have human-created maps with villages that follow these rules because they have been carefully planned out. Other games are designed to have the player build villages and cities that are as realistic as possible, such as *Cities: Skylines*[8]. Even though it is not mandatory to follow these principles in this game, it is very useful to do so. What is difficult, however, is to create a game in which villages and cities are automatically generated following these rules. We find that the villages in Minecraft do not do this, and we want to use the terms introduced above to show specifically why these villages do not look natural.

Let us start with the first point, the village center. Each village in Minecraft has not one, but several centers, each of which is defined as a subchunk (a 16-by-16-by-16 large section of the world) in which there is at least one claimed bed, bell, or job site block. In the game, the village is then defined as the union of all these centers with the 26 subchunks surrounding each center [25]. In addition, as explained in Section 2.3, the selection of houses is purely random. As a result, buildings with functions are sometimes generated far away from the

[8]Official website: `https://www.paradoxinteractive.com/games/cities-skylines/about/`.

lively part of the village, instead of where the most traffic is. This means that there is no clear village center at all, at least not by the above definition. The houses are distributed in a way that does not have an obvious order or plan, as can be seen in Figure 2.9.



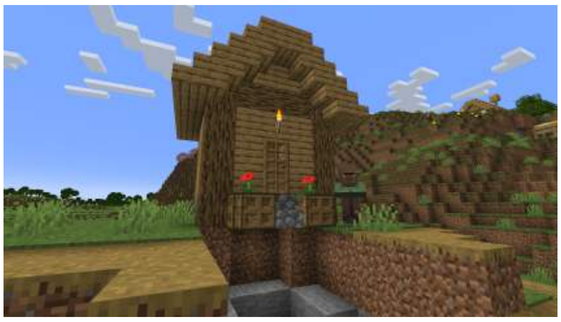
Source: Author's image.

Figure 2.9.: The eight houses of this village are randomly distributed throughout the village. There are only two buildings at the crossing with the tree in the middle, where we could imagine a village center. Moreover, the only house with a function, a fletcher house (plus-shaped roof at the very back), is at the very end of one of the paths and thus far away relative to the overall size of the village for the villagers living in the two houses at the bottom left of the picture.

Villages in Minecraft do not follow the second point either. Although they are usually smaller than villages and cities in real life, we would expect at least a bit of road hierarchy and more structure. However, as can be seen in Figure 2.9, for example, all roads have the exact same width, regardless of what function they serve. In addition, there are paths that extend beyond the boundaries of the village and thus have no function at all, like the path left to the fletcher house in Figure 2.9.

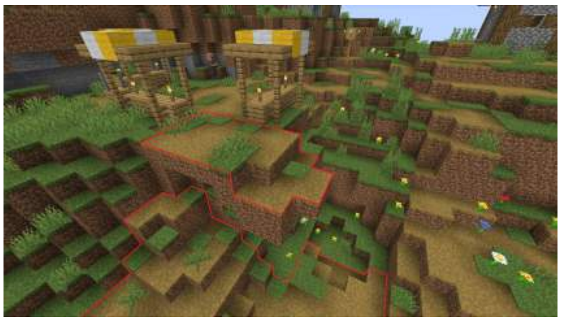
There is another problem with the villages in Minecraft, for which we have not mentioned any connection to the real world yet, because this is a natural thing which we usually take for granted. When a road is planned and built in the real world, the characteristics of the local terrain are taken into consideration. A good example of this is a slope. The most natural way to overcome a slope whose steepness is beyond the capacity of person, animal, or machine is a hairpin turn, that is, a snake-shaped line at whose turning points are the bends [29]. It would be difficult, if not impossible, to take the direct, straight path across the slope, even if it is the shortest. Even in less extreme examples, roads often follow the terrain in an attempt to minimize the amount of efforts required, regardless of the type of locomotion. For a human, this effort is the energy needed for muscle movement, and for modern vehicles, for example, it is the torque of the engine.

Villages in Minecraft, on the other hand, do not follow this principle at all. Every path is perfectly straight and does not respect the terrain it runs over. This often leads to generated paths that are difficult (i.e., only with a lot of jumping) or even impossible to traverse (see Figure 2.10). Although the world of Minecraft is made of blocks, it is still possible to build sloped or curved paths, but this is simply not done.



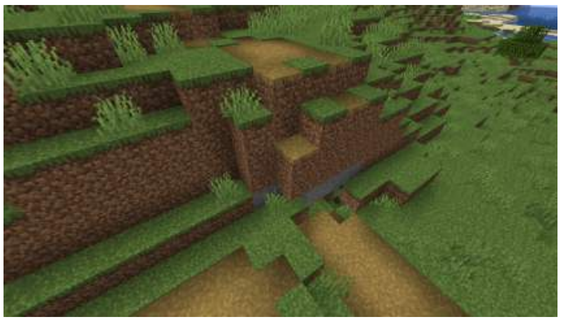
Source: Author's image.

(a) An unreachable house.



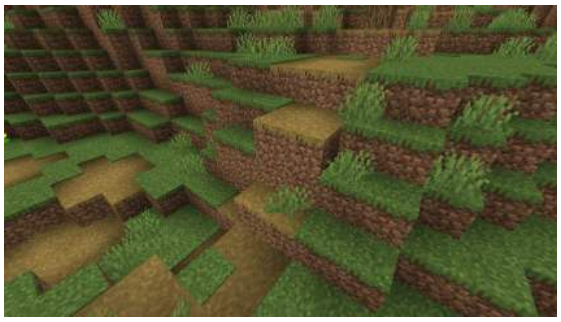
Source: Author's image.

(b) A path running over a big cliff.



Source: Author's image.

(c) A path running over a small cliff.



Source: Author's image.

(d) A path on a steep slope.

Figure 2.10.: Examples of paths generated in such a way that it is impossible to traverse them. All these pictures were taken within the same village. Since there are many paths in Figure 2.10(b), the important path is marked in red.

At this point, one important thing must be said. The villages are almost completely functional despite these flaws. They do not need a village center or road hierarchy. Even the questionable generation of paths is not a problem, because there is (almost) always a way to get to the destination without using the existing path. For the player, this is even less of a challenge, because unlike the villagers, they can simply mine and place blocks, and thus always make a path. Rather, all of these negative points are related to the general aesthetics of the village. They just do not look the way we would imagine a village to look in the real world. This fact can lead to a decrease in the player's immersion in the game, as they do not feel like they are in a real village, but just a collection. of randomly arranged buildings without any thought-out structure.

This is exactly why we want to generate more natural-looking villages in Minecraft in this thesis. As stated in chapter 1, it is not mandatory to use Minecraft for this research. However, we can now see that Minecraft, with its already existing villages, provides a good base which we can build on and try to generate better villages.

2.3.2. The Generative Design in Minecraft competition

We are not the first ones who had this idea. Since 2018, the so-called *Generative Design in Minecraft (GDMC) AI Settlement Generation Competition*[9] has been held every year. The goal is to design an algorithm that generates a settlement (i.e., a village) in any Minecraft world by placing and removing blocks without human supervision. After the algorithm is submitted to the contest, it will be applied to some previously unknown worlds, and

[9]Official website: `https://gendesignmc.engineering.nyu.edu/`.

the resulting villages will be evaluated by experienced judges based on certain criteria (see below). The individual or team that designed the algorithm with the best average score wins [32]. The winner is announced at the annual IEEE Conference on Games.

There are four categories of criteria by which a village is evaluated.

Adaptability. Judges evaluate how well the village adapts to the terrain and environment.

Functionality. This includes, for example, whether the village is safe from hostile mobs, whether the village and everything in it is accessible to the player, and whether the village assigns functions to the houses and the villagers living and working in them.

Believable and evocative narrative. For this, the judges ask themselves if the village tells an interesting story. What this means is whether the function of the village is clear (fishing village, farming village, etc.) and whether that function is well realized. For example, a fishing lodge should be near a water source and a farm should be surrounded by fields.

Visual aesthetics. For this point, the judges evaluate the general appearance of the village, such as the appearance and variety of buildings or the existence of unrealistic structures [32].

In fact, we submitted our algorithm to the GDMC competition 2023, although an incomplete version of it. For the results, see Section 5.3. In our algorithm, we focus mainly on the first two points. We mentioned that paths should adapt to the terrain, which belongs to adaptability. For the second point, we take care that all buildings are accessible (this is closely related to adaptability) and that the houses have a clear function. We do not value the remaining points a lot, however, we indirectly implement them at least a little bit. The third point is partially fulfilled by the fact that it will be recognizable how the village has grown, and the fourth point is partially fulfilled by the fact that we design houses that look better than the default ones in Minecraft.

2.4. Amulet Editor

The algorithm we will present in Chapter 4, as indicated in Chapter 1, works with matrices and not with a Minecraft world directly. However, since most of these matrices contain information about the Minecraft world, we obviously need a way to obtain this information. Since Minecraft is such a popular game, a lot is known about the way worlds and other data are stored. Moreover, in the large community of Minecraft, there are already some individuals who have been working on the question of how to extract this information and convert it into a simple format.

The software we will use is the *Amulet Editor*[10], or simply *Amulet* for short, a open-source Minecraft world editor with a visual interface. Particularly noteworthy is the user's ability to place, remove, or replace almost arbitrarily large selections of blocks, and to copy between different worlds (regardless of the game version or platform[11]), all within this interface [13]. The most interesting feature

Source: `https://pbs.twimg.com/profile_images/1384482182694010883/pt6sDBOn.jpg`.

Figure 2.11.: The logo of the Amulet Editor.

[10]Official website: `https://www.amuletmc.com/`.
Github: `https://github.com/Amulet-Team/Amulet-Map-Editor/`.

[11]Amulet supports all versions starting from Java 1.12 and Bedrock 1.17.

for us, however, is the API Amulet-Core[12] which provides the main functionality of the editor. The core is written in *Python*[13] and converts the information into a custom format that is easy to work with, which allows us to manipulate the world data outside the interface with Python scripts [13]. In addition, we can easily place the generated village in the actual world with Amulet by passing the coordinates of the houses and paths from our generated village to Amulet and telling it to place (or remove) blocks in the specified locations within the world.

[12]Github: `https://github.com/Amulet-Team/Amulet-Core/`.
[13]Official website: `https://www.python.org/`.

3. Related work

There are many terms and research areas that are closely related to this thesis. In particular, the two terms “procedural generation” and “ant colony optimization algorithms” from the title of this thesis need to be carefully defined. Such related terms will be the topic of this chapter. In addition, we will examine existing methods for village generation both in general and in the game Minecraft, and we will emphasize the main differences between other approaches and our approach.

First, procedural content generation is introduced in Section 3.1. Sections 3.2 and 3.3 explain the concepts of pathfinding and agent-based simulation. Then, Section 3.4 presents ant colony optimization algorithms in detail, and finally, Section 3.5 examines some existing academic work on Minecraft.

3.1. Procedural content generation

The first term that is defined is *procedural content generation* (*PCG* for short). PCG is the creation of game content by algorithms, with little or only indirect input from the outside [36]. In other words, PCG is computer software that can create game content all by itself, or in collaboration with humans. We define “content” as certain things that can be found in a game. This includes, for example, levels, maps, textures, items, weapons, or characters, but also things that the player cannot touch directly, like stories, quests, or music. A “game” can mean many things, such as video games or board games. Regardless of the type of game for which content is generated, that content must be playable. That is, for example, the player should be able to complete a generated level or fulfill a generated quest. The terms “procedural” and “generation” mean that some kind of computer procedure generates something. A PCG method can be (and usually is) executed on a computer and provides an output [34].

At this point the question arises why PCG is used. Probably the most obvious reason is that it saves on human designers or artists, who are expensive and slow. Today’s games usually require a lot of manpower, and often games are developed by hundreds of people over the span of years. With the help of (good) PCG algorithms, a company could replace some of the designers and artists, thus saving money and time. Another reason, for example, is that PCG methods often generate more creative content than a human would. Humans tend to imitate others and themselves. An algorithm, on the other hand, may generate content that is completely different from what a human would design, and still be a valid and functioning solution for its intended purpose [34].

The game Minecraft itself makes great use of PCG. As indicated in Section 2.2, the entire worlds of Minecraft are procedurally generated. The seed of a world, which is used as the seed for the random procedures when generating a new world, can take on a total of $2^{64} = 18,446,744,073,709,551,616$ different values, which means that this is the number of unique worlds that can be generated by Minecraft [25]. The probability that a player, when entering a new world, ends up in the same world as before is approximately $5.412 \cdot 10^{-18}\%$. Moreover, villages as introduced in Section 2.3, are procedurally generated. This means that it is virtually impossible to find the same village twice (within in the same world or in two worlds with different seeds).

At this point we can also see exactly how our algorithm fits into the PCG category. The game for which content is created is Minecraft, and the content is villages. The algorithm is run by a computer and produces an output, which is the villages. Thus, our algorithm meets all the conditions of the above definition and belongs to PCG methods.

In general, we can say that the generation of villages and cities with a computer is a PCG method. There are already some proposed methods for this. In [15], Greuter et al. present a method to generate cities in real time. The buildings are procedurally generated and their shape depends on their position within the city. The road network on the other hand is completely static and nothing more than a perfect grid of roads on a flat ground. Thus, unlike in our approach, paths or roads are not dynamically generated in [15], but the houses are. Parish et al. [27] go one step further and also generate the road network using L-systems. For this purpose, they present several types of road networks, one of which differentiates the type of road according to the slope of the terrain. The different types of roads also form a hierarchy with clearly identifiable major and minor roads. However, the roads are not generated with respect to the terrain and do not further adapt to it (except that they are wider or narrower depending on the slope). We, on the other hand, want our approach to generate roads that fully adapt to the terrain.

This is exactly what Kelly et al. implement in [21]. First, they generate a main network of streets, i.e., major roads, between nodes on the map that must be placed manually. The user determines which nodes are connected, and the algorithm then generates a road between such nodes, trying to keep elevation as small as possible. Then, as in [27], the areas enclosed by major roads are filled with minor roads using L-systems. Finally, similar to [15], houses are procedurally generated and placed between the roads. The biggest difference to our algorithm is that in [21] the places where major roads meet have to be determined manually. In our approach, however, the entire path network is generated automatically.

There are two other major differences between all three proposed methods and ours. First, no explicit functions are assigned to the individual buildings. In [27], buildings have different heights depending on the population density of the area in which they are constructed, but no further meaning. Kelly et al. [21] divide houses into down-town, suburban, and industrial buildings depending on the city district they are built in. However, within these three categories, no building plays a different role than another. We aim to improve this by assigning each house a specific function that is clearly identifiable from its appearance and position within the village.

The second difference is that none of these cities simulate natural growth. The cities of Greuter et al. [15] are generated in real time and are theoretically infinitely large, but because of their primitive structure we cannot speak of natural growth. They simply attach a new square to an old square of the same size. In both the methods of Parish et al. [27] and Kelly et al. [21], growth is simulated by the L-systems. However, the goal of growth in [27] is to start generating major streets first and over time simultaneously fill the areas between them with minor streets. This is similar in [21], where only the minor roads between the main roads grow. Neither case simulates that the city grows larger over time, which can cause streets to change their function or sometimes even their entire course. We want to implement this. In our method, a village grows, and with each growth cycle, the already existing path network is expanded and adjusted, if necessary.

3.2. Pathfinding

In our algorithm, we later want to find paths between houses to generate a path network. This process can generally be called *pathfinding*. The goal of a pathfinding algorithm is usually to find a[14] shortest path between two points. This type of algorithm is of great importance in

[14]Note that we deliberately say “a” and not “the” here since the shortest path between two points is not necessarily unique.

video games, because NPCs (non-player characters) often use it to move around in the world of a game. Among the most widely used pathfinding algorithms are the A* search algorithm by Hart et al. [18] and Dijkstra's algorithm [6]. However, these two algorithms and many other pathfinding algorithms are, as said, methods to find a shortest path. This is not what we want to achieve. As explained in Section 2.3.1, we want the paths of the village to be natural, and the shortest path is not always the most natural.

Galin et al. [11] asked themselves the same question and developed a method to procedurally generate natural roads. In their algorithm, a road is generated between two points that adapts to the characteristics of the terrain. For this purpose, a shortest path between the two points is found which at the same time minimizes a cost function. For the shortest path the A* algorithm is applied. A number of factors are taken into account in the cost function. For example, a road should follow the slope of the terrain and adapt to natural obstacles such as forests, rivers, or lakes. A road should also not make too sharp turns, and bridges and tunnels should be constructed only where it makes sense, since this is an expensive and time-consuming matter in real life.

The ideas in [11] are very similar to ours. Like in our approach, they search for natural roads and define naturalness according to similar criteria as we do. Through their algorithm, roads are generated that adapt to the terrain with many curves and even form the hairpin turns mentioned in Section 2.3.1. However, only a single road is generated, while we are going to generate an entire network of paths. We think that the most natural course of a road is different when there are more roads and more start and destination points. Unlike Galin et al. [11], we want to investigate this in more detail.

3.3. Agent-based simulation

Another term related to the topic of this thesis is *agent-based simulation.* It is the concept of many small entities (agents) having decision-making or action capabilities. Ant colony optimization algorithms, presented in Section 3.4 below, are an example of a agent-based simulation. An agent is an individual consisting of certain properties and rules that determine its behavior and decisions. The agent lives in an environment where it interacts with other agents. These interactions also follow certain rules. However, an agent can also act alone in its world. The agent has a goal that it pursues, and it has the ability to learn and change its behavior over time [23]. The goal of agent-based simulations is that of emergence, which means that new properties or structures are formed from the interaction of smaller elements (i.e., the agents). Our algorithm belongs to agent-based simulations. The villagers that will be simulated later form the agents, and the Minecraft world is the environment. Moreover, the villagers follow certain rules when they move around, and they all have the goal of finding a natural path from their own house to another. Villagers also learn since they evaluate and remember paths they have taken before in order to possibly traverse them again in later iterations. The goal of our villager simulation is for a natural network of paths to emerge from this.

In [35], agent-based simulation is used to generate a town that grows as in our algorithm. For this, the life of the individual inhabitants of the town is simulated. They follow a simple routine of consuming food, working, trading, and resting. The agents act on a map of nodes over which they move. Depending on how much a node is visited or what actions are performed on a node, it is transformed into, for example, a street or a house. Thus, a road hierarchy is formed. What makes this method similar to ours is that the type of road depends on how much it is traversed. However, roads are formed without any regard to the slope of the terrain, and houses have no functions. As mentioned earlier, these are two things we will implement in our algorithm.

3.4. Ant colony optimization algorithms

In this section, we define the last unexplained term from the title of this thesis. The topic of this section is *ant colony optimization (ACO) algorithms*. First introduced in 1996 by Marco Dorigo et al. [7], they are inspired by ant colonies in real life. The main idea is to imitate the way ants cooperate to solve a common goal, especially when they are foraging for food. The ants' behavior is based on indirect communication using so-called *pheromones*. These are a chemical substance that the ants can secrete[15] and detect through their sense of smell. As the ants walk from their colony to a food source and back again, they leave such pheromones on the ground, creating a pheromone trail. Other ants smell these pheromones and tend (that is, decide probabilistically) to choose a path where there is a strong concentration of pheromones. The shorter a path is, the more often ants can traverse it, as it simply takes less time than traversing a long path. This means that ants can release pheromones more often on a short path than on a long one. It follows that the pheromone concentration increases faster on a short path, which means that more and more ants tend to choose this path and in turn release even more pheromones. In addition, the pheromones evaporate[16] over time. So after a while, the ants forget, so to speak, paths that they have walked before, and if a path is not traversed regularly, it disappears. Since a short path used by many ants does not easily evaporate, at some point almost all ants follow such a short path to the food source [8].

ACO algorithms are usually applied to graphs. The exact procedures can best be explained by an example, for which the traveling salesman problem is usually used. Given a list of cities and distances between them, the goal is to find the shortest possible path that visits all cities exactly once and returns to the starting city. In [8], many different implementations of ACO algorithms are presented that solve this problem. However, we will focus on the method originally presented in [7] since it has the most impact on our algorithm.

Let $G = (C, L)$ be a graph where the set L of all edges fully connects the nodes C. In [7], $m \in \mathbb{N}$ ants simultaneously construct a route for the traveling salesman problem. For this, the ants are placed in a randomly chosen city, and in each construction step, an ant $k \in \{1, \ldots, m\}$ applies a probabilistic rule to choose the next city on its route. Let $i \in C$ be the city where ant k is currently located, let $\mathcal{N}_i^k \subset C$ be the neighborhood of ant k in city i, i.e., the set of all cities that ant k has not yet visited, and let $\alpha, \beta \in \mathbb{R}_{\geq 0}$ be two parameters. The probability that ant k chooses a city $j \in \mathcal{N}_i^k$ is defined as:

$$p_{ij}^k := \frac{(\tau_{ij})^\alpha (\eta_{ij})^\beta}{\sum_{l \in \mathcal{N}_i^k} (\tau_{il})^\alpha (\eta_{il})^\beta}. \tag{3.1}$$

Here, τ_{ij} is the pheromone level on the edge between city i and j, and $\eta_{ij} := 1/d_{ij}$ is a heuristic that favors nearby cities (d_{ij} is the distance between city i and city j). The higher τ_{ij} and η_{ij} are on the edge between i and j, the more likely an ant is to go to city j. When $\alpha = 0$, the closest city is more likely to be selected, and when $\beta = 0$, ants are influenced only by pheromones. An ant k has a memory $\mathcal{M}^k$ in which the sequence (i.e., the ordered set) of all previously visited cities is stored. Using this memory, the neighborhood $\mathcal{N}_i^k$ from (3.1) and the length $C^k \in \mathbb{R}_{\geq 0}$ of the tour $T^k \subset L$ in (3.4) below are computed.

There are two approaches to implement the path search. Either all ants search in parallel, which means that in each step all ants go to only one city, or they search sequentially, which

[15] According to the Cambridge Dictionary, "to secrete" describes the process of an animal or plant producing and releasing a fluid: `https://dictionary.cambridge.org/dictionary/english/secrete/`. Synonyms are "to give off" or "to emit".

[16] According to the Cambridge Dictionary, "to evaporate" describes a liquid to change to a gas, that is, to dissolve into thin air: `https://dictionary.cambridge.org/dictionary/english/evaporate/`. Synonyms are "to disappear", "to dissolve", or "to fade".

means that it is not an ant's turn until the previous ant has found a complete route. It is mentioned in [8] that the type of implementation does not make much of a difference, and we will choose the sequential method later in Section 4.4.

After all ants have constructed a tour, the pheromone trails are updated. First, the pheromone value on each edge of the entire graph is decreased by a constant factor to simulate pheromone evaporation. Let $\rho \in (0, 1]$ be the pheromone evaporation rate. For all $(i, j) \in L$, pheromone evaporation is simulated by

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij}, \tag{3.2}$$

where ρ prevents an unlimited amount of pheromones from accumulating on an edge, and causes previously traversed but poor routes to be forgotten. If an edge is not used by ants, the pheromone level on it decreases exponentially.

After evaporation, all ants secrete pheromones on the edges they have traversed. To be precise, for all edges $(i, j) \in L$, the pheromone level τ_{ij} is adjusted by

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}, \tag{3.3}$$

where $\Delta\tau_{ij}^{k}$ is the amount of pheromones that ant k secretes on edges it has visited. This amount is defined as

$$\Delta\tau_{ij}^{k} := \begin{cases} \frac{1}{C^k}, & \text{if } (i, j) \in T^k, \\ 0, & \text{otherwise,} \end{cases} \tag{3.4}$$

where C^k is the length of ant k's tour T^k as mentioned above, that is, the sum of the lengths of the edges of tour T^k. Therefore, according to (3.4), the shorter the found route is, the more pheromones are secreted. On edges that are used by more ants and are part of shorter paths, more pheromones accumulate, and they are therefore traversed by more ants over time. This leads to the ants converging to the shortest route, i.e., the solution to the traveling salesman problem.

Even though ACO algorithms are usually understood as an optimization method, there are already some attempts to use them for PCG. In [1], ACO algorithms are used to generate procedural animations. For example, if the desired animation is to make a human-like object jump, ants search for a sequence of motions that optimally satisfy this goal. In [30], the ants generate a maze by moving across a grid and looking for the exit on one side of that grid. Their path choice defines where walls are placed and where they are not. Saraiva et al. [33] use ACO algorithms to generate maps for the board games Terra Mystica and Settlers of Catan. To do this, the ants move around the board looking for an arrangement of terrain tiles that is as balanced as possible. However, we are not aware of any existing method that uses ACO algorithms to generate villages or cities.

3.5. Academic works on village generation in Minecraft

Because Minecraft's worlds have a very simple block structure, the game is a good way to visualize ideas and concepts. Instead of creating and viewing complex structures using rendering software or game engines, Minecraft offers the possibility to have access to a final product faster, which only comes at the cost of less detail. When it comes to automatically generating these ideas and concepts, Minecraft provides a great opportunity to use tools like Amulet (see Section 2.4) to view the generated objects in a simple and accessible way, all while being within an interactive environment. Especially in the field of PCG, many researchers have chosen to use Minecraft in recent years. In the context of the GDMC competition, many new methods have been developed using Minecraft as an environment to test and visualize new algorithms. In this section, we will therefore briefly examine a few of the academic works on PCG in Minecraft and specifically on the generation of villages in Minecraft.

Beukman et al. [3] use a neural network to generate villages in Minecraft (and other structures in other games). For this, a generator is trained to create a two-dimensional tilemap, where each tile has a function such as house, street, or garden. During generation, a fitness function is maximized, taking into account factors such as the proportion of different types of tiles. Each of these tiles is then subdivided into smaller tiles by another generator, each representing a column of blocks. Depending on the type of large tile, a different fitness function is maximized. For example, a house should consist of walls on the outside, be hollow on the inside, and have a roof. The resulting village has an interesting structure, but lacks any details. The buildings are nothing more than large, empty cuboids and the streets take up the entire area between houses tiles, which makes the village look very lifeless overall. On top of that, the cities have to be generated on a perfectly flat ground, which is not the case in a normal Minecraft world.

The next algorithms will all be methods for generating villages in Minecraft, and have been developed for and in some instances submitted to GDMC competitions in recent years. While there are quite a few academic papers on this, we feel that most of the results are rather bad. This is also reflected in the rankings of these algorithms in the competition, with most of them having scored low. Nevertheless, we want to present three of them, since they have similar ideas to ours. All three papers use agent-based simulations.

The first method is [5], which was developed as part of a future submission for the contest (the method has not yet been used in a submission for the GDMC competition). First, the region is divided into squares of the same size, where a square must be accessible from any other square by normal movements in Minecraft (that is, without having to mine or place blocks). Then, inhabitants are instantiated, following a simple behavior. Depending on whether they are hungry or want to sleep, for example, they perform different actions. If they want to sleep but have no house, they build themselves a shelter on one of the previously defined squares that has not yet been occupied by another resident. Villagers also reproduce, which allows the village to grow over time. Although Christiansen et al. [5] do not explicitly mention it, the results suggest that the paths between houses are perfectly straight lines that minimize the Chebyshev distance[17] between them. What we like about the approach is that the village can grow and all houses can be reached from any other house within the village. Moreover, the agents have meaningful interpretations as they represent the actual villagers. However, as in the default villages in Minecraft, the paths are just straight lines (although also diagonal in this case) that do not adapt to the terrain at all. Also, there is no building that has any function other than that of a residential house.

In [10], a village is generated by a number of agents, three of which we will look at in more detail. The first agent is responsible for generating roads. Starting from a predefined initial road, it begins a random walk. As soon as it moves too far away from the existing road network, it connects its current position with the closest existing road. After that, another agent traverses the generated network. If it finds two points that are close to each other but the distance between when going via the road network is very long, it connects these two points with a new road. Next, a last agent wanders over the entire region, looking for areas between the generated roads that have enough space to place a house. Finally, houses are placed on these areas. We think that the ideas behind the generation of the road network with agents are good, and we find that the network formed connects all parts of the village with meaningful and organic paths. However, the paths do not respect the slope of the terrain and sometimes even include non-traversable parts. Besides, none of the houses

[17]This means that paths run either parallel to the block grid of the world or at a 45-degree angle to it. The distance between a block and all its surrounding eight blocks is considered to be the same.

has a function.

The last algorithm we want to discuss is AgentCraft by Iramanesh et al. [20]. Here, the Minecraft world is masked with a grid of nodes, and at the beginning a road with a house next to it is placed on one such node. Agents simulating individual villagers are also placed and, as in [5], follow simple rules that determine where in the village they move to. The more often they walk over a certain node in the world, the more its value increases. When this value exceeds a certain limit, the node is connected to the existing road network by a new road. For this, an attempt is made to place a straight path from this node to the closest node of the road network. If this generates a path that cannot be traversed, the path is split into two smaller paths, or the algorithm searches for a path to another node of the network. If all fails, no path is generated. Agents reproduce and build more and more houses over time, thus gradually growing the village and connecting new parts to the existing road network (if possible). We think that the villages generated by [20] look great. The roads do not just run parallel to the block grid and there are houses with different functions. This is also reflected in the performance at the GDMC competition, where the algorithm won second place in 2021. Still, there are some things we would improve about it. For example, it can happen that two buildings that are close to each other are far away from each other when walking between them on the road network. Also, some buildings may not be connected to the road network at all.

In addition, there are two concepts that we defined in Section 2.3.1 as being natural in a village, both of which are not implemented by any of the methods. First, none of the villages have a clearly identifiable center, and second, none of the road networks follow a hierarchy.

4. Methodology

This chapter is the main part of this thesis and is structured as follows. We will first summarize the motivation for our research and precisely define our goal. Then, we will present the algorithm we designed from Sections 4.1 to 4.6, also going into detail about the motivations and rationale behind our decisions.

We have already explained the most important points of our the motivation for this research in section 2.3.1. We think that the villages in Minecraft do not look natural, and we have given specific points as to why we think so. The two biggest complaints were the placement of the houses within the village and the course of paths between them. We believe we can generate villages that solve both of these issues. Since this is a very vague formulation of what we want to achieve and, more importantly, how we are going to achieve it, we now define exactly our goal and the techniques we will use to achieve it.

We want to design an algorithm that generates a village in an arbitrary Minecraft world within an arbitrary area[18]. The only input needed from the user for this is the world and a bounding box where the village is to be generated. For many other parameters we give default values, however, these can also be adjusted according to the user's needs. For example, the user can specify how many houses should be placed, how long life in the village is simulated, or how often the village grows. After that, the village is generated without any further interaction from the user and is also automatically placed in the world.

The generated village should be free from the deficiencies we mentioned in Section 2.3.1. To be more precise, houses should not be distributed seemingly randomly over the village. A clear village center should be recognizable, in which the settlement density is highest, and further outside houses should be placed rather sparsely. The placement of buildings with functions should be in locations that make sense and should not contradict their very purpose. A farmhouse, for example, should be placed in the outskirts, where there is more space for fields. Furthermore, the paths that connect the houses should respect the terrain and follow a road hierarchy. A path should be traversable in any case and not be too steep, as this would not follow the principle of paths in real life. Clear major paths should be recognizable and distinguishable from minor paths by being more developed.

To achieve this goal, we want to proceed as follows. Placing houses succeeds by simple methods. They are placed in locations that are as flat as possible and as close as possible to the village center, which is defined as the average x- and z-position[19] of all houses in the village (see Sections 4.1 and 4.3). Functions are assigned to houses only at the end of the algorithm, depending on their position in the village (see Section 4.6). For generating paths, we want to use ant colony optimization algorithms from Section 3.4. The basic idea behind this approach is that ants find a path through the village whenever a villager wants to go from one building to another. Over time, the pheromones left by the ants form a network of paths connecting all the houses. Paths with more pheromones form major paths, which are built wider than minor paths. To obtain the desired properties, the ants follow certain rules during pathfinding. For the details, see Section 4.4.

[18]To be precise, this only includes suitable areas. This should be intuitive, however, as it makes no sense to place a village in the middle of an ocean without a solid base, for example. For more detailed information, see Section 4.1.

[19]Note that the vertical component in Minecraft is the y-coordinate.

Overview

We will now present an overview of the algorithm that automatically generates the villages. Since it is quite complex, we will divide it into smaller steps and explain them one after the other. The order we choose for this is exactly the order in which these steps are actually executed. While this leads to less important parts being explained earlier than more important parts, it makes it easier to understand the design decisions of the algorithm. The main algorithm can be seen in Algorithm 1. As mentioned earlier, the only input is the Minecraft world in which the village is to be generated, and a bounding box that determines in which area the village is to be generated. The algorithm does not output anything, but instead manipulates the world directly.

```
input: A Minecraft world, and a bounding box.
output: None.
1  load the world and prepare it;
2  while time limit is not reached do
3      if first iteration then
4          initialize the village;     // transform the world into data we can
                                          easily work with and extract important
                                          information
5      end
6      else
7          update the village;         // update the world's and village's data,
                                          taking already existing houses and
                                          paths into account
8      end
9      populate the village;           // place houses
10     simulate life;                  // generate or update the paths by
                                          simulating ants
11 end
12 build the village;                  // build the houses and paths in the
                                          Minecraft world
```

Algorithm 1: The main algorithm.

Let us take a closer look at the steps of Algorithm 1. First, we load the world using Amulet and extract the data we need. Moreover, we also need to edit the world a bit. To be precise, we need to remove all the trees inside the bounding box. For the reasoning behind this, see Section 4.1.

We then enter a while loop that executes three instructions. First, we update all variables of the village or initialize them if this is the first iteration. We introduce the initialization in Section 4.2, whereas the updating is introduced in Section 4.5, since the steps that follow are necessary for updating. Next, we populate the village, that is, we place houses in it, as described in Section 4.3. In our examples, we place eight houses at once and a villager lives in each of them. Finally, we simulate life in the village, generating the paths between the buildings. For this, the villagers are simulated by four ants each, which find a way through the village. This is the most important part of the algorithm and is described in detail in Section 4.4. The reason that these instructions are in a while loop is that the village can grow in this way. If we keep placing the same number of houses as before and generating paths between all of them, the village can simulate natural growth, as we will explain in Section 4.5. The while loop runs until a time limit is reached. The motivation for this is that the GDMC competition limits the running of the algorithms to ten minutes, and we want to guarantee not to exceed this limit[20]. We could also easily use a for loop with a fixed number of growth iterations instead if we want to control exactly how often the village grows.

After the while loop, we build the village. Up to this point, we have only manipulated the world once in the very first step, and all other simulations have not interacted directly with the world. In this step, however, we are indeed placing blocks into the Minecraft world. The type of house (its function) and width of the path we build depends on a few factors, as we explain in Section 4.6.

4.1. Loading, analyzing, and deforesting the world

The very first actual step is to load and analyze the world, and additionally clear it of all its trees. An overview of this can be seen in Algorithm 2.

```
  input: A Minecraft world, and a bounding box.
  output: The heightmap, watermap, and lavamap of the region.
1 load the world;
2 extract heightmaps from the world;
3 find trees, water, and lava;
4 remove trees and mark water and lava;
```

Algorithm 2: Loading, analyzing, and deforesting the world.

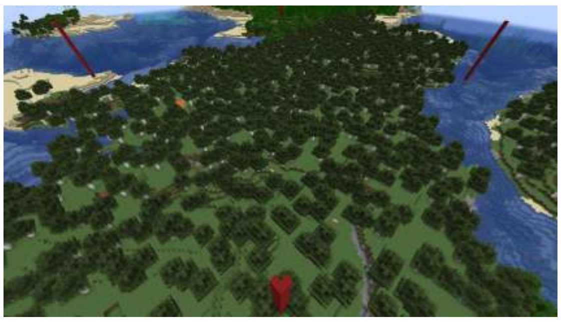
First, we load the Minecraft world with Amulet. This gives us access to important data in a format that we can easily work with. To be precise, we are interested in the so-called *heightmaps* of the world, which we extract in the second step. Heightmaps are maps that store the y-level of the highest block of each coordinate. A Minecraft world has different types of heightmaps. This allows us to find all the trees, water blocks, and lava blocks in the region, and then remove those trees and save the positions of water and lava. For details, see Appendix B. In the end, we output a new heightmap, which is now updated for the region without trees.

Before we give an example, we should explain why we remove trees and save the positions of water and lava. The problem with trees is that they take up a lot of space and thus get in the way for some of the steps that follow later. For example, trees block possible locations where we could place houses. For placing houses, we will look at how level the terrain is. This is done in Section 4.3 by using the heightmap generated by Algorithm 2. If trees were part of this heightmap, they would create a lot of "noise" and might make the heightmap appear uneven and rough despite the ground being actually level. Of course, this is not the most natural approach because trees in a village are cut down only when there is an immediate need to do so. A different approach would be to just remember the position of trees and update the heightmap, and remove trees only when we actually want to place a building at this position. This is problematic, however, because it creates a high probability that we will only partially remove trees and leave half-destroyed trees in our village. In Minecraft, trees often grow into each other and, since there are many different types of trees, it is unfeasible to use an algorithm to remove only one of these trees and leave the others untouched. In fact, we encounter exactly this problem at the edges of the region. We will often find trees there that are half cut off. However, as we said, there is no trivial solution for this, so we ignore this problem.

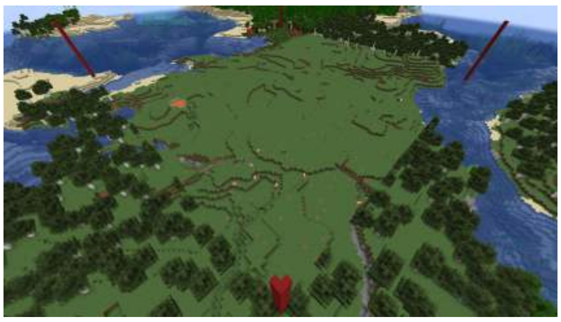
[20]The exact rule is that the algorithm should not run substantially more than ten minutes. Since the instructions in the while loop are very costly, we may enter the loop just before the ten-minute mark, but not finish the iteration until two or three minutes later. However, this is within the scope of GDMC competition.

The reason for saving the water and lava positions also has to do with placing houses, but it is simpler. Intuitively, a house should not be placed on water and especially not on lava. In addition, we treat these two liquids separately, as buildings should keep an extra distance from lava. Blocks that are too close to lava can burst into flames, which would be fatal especially for houses made of wooden blocks.

Example of Algorithm 2. With the following example of Algorithm 2 we want to illustrate the difference between a region with trees and one without. For simplicity and consistency, we will use the same world for demonstrating this algorithm and the following algorithms. The algorithm gets as input a Minecraft world[21] and a bounding box. We specify the bounding box with the x- and z-coordinates of the two corners that are furthest and closest to the origin of the world ($(x, z) = (0, 0)$), respectively. For example, $(x_{\text{lower}}, z_{\text{lower}}) = (-213, 284)$ and $(x_{\text{upper}}, z_{\text{upper}}) = (-43, 454)$, resulting in a bounding box of dimensions 171-by-171 (all bounds are inclusive). Note that the box does not necessarily have to be square. We then load this region of the world[22]. Figure 4.1 shows the region before and after running the algorithm. The exact characteristics of the terrain can be seen only after the trees are removed, and we can see that it is a very flat region with many areas suitable for placing houses. The edges of the region can be clearly made out, as the trees there are cut abruptly, leaving partially destroyed trees. The effect can be seen even better in Figure 4.2, which shows a heightmap once before and once after the execution. We can easily see how much space the trees actually take up and how many possible places for houses they block as a result. The water and lava maps of this region are shown in Figure 4.3.


Source: Author's image.

(a) The region before tree removal.


Source: Author's image.

(b) The region after tree removal.

Figure 4.1.: The world's region we have defined by the bounding box. The red columns show the corners of the box.

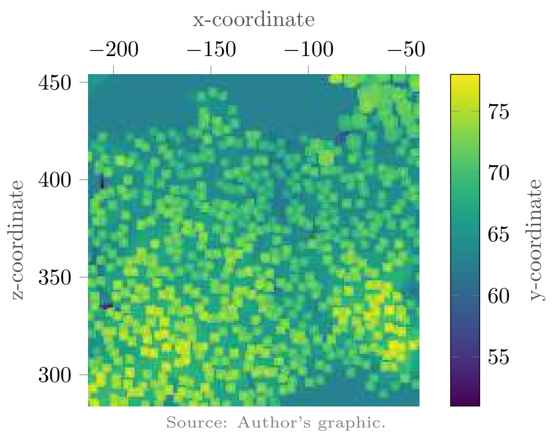
4.2. Initializing the village

In order to start placing houses in the village, we must first make some further preparations, which are summarized in Algorithm 3.

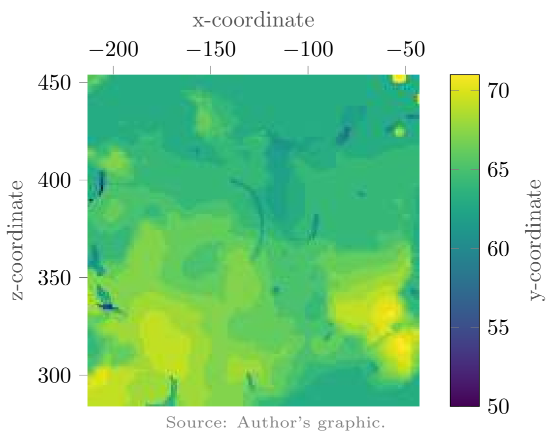
In the first step, all connected areas are found. Two blocks belong to the same connected area if, and only if, we can move from one block to the other while obeying two rules:

1. At any time, we can only go exactly one block in one of the four cardinal directions (i.e., in the von Neumann neighborhood of a block, see Figure 4.4).
2. We cannot go to a block if it is more than one block below or above the current block.

[21]On Windows, the Minecraft folder is usually found in `C:\Users\user\AppData\Roaming\.minecraft\`, on macOS in `~/Library/Application Support/minecraft/`, and on Linux in `~/.minecraft/` [25].
[22]The seed of this example's world is -6686748516404217046.

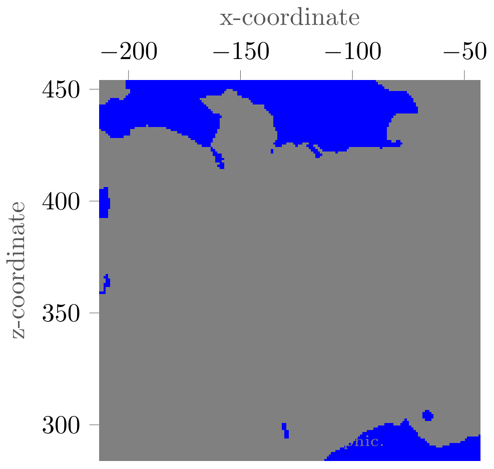


(a) Heightmap before tree removal.

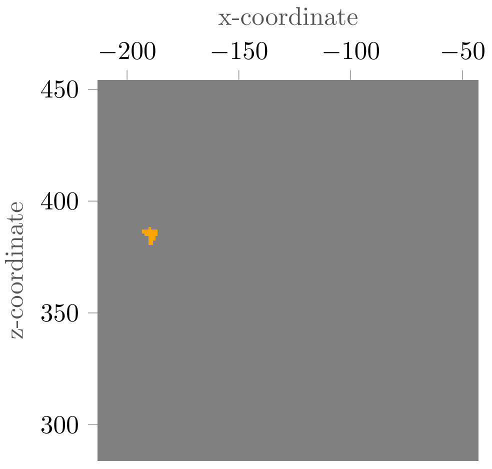


(b) Heightmap after tree removal.

Figure 4.2.: The heightmap of the region before and after Algorithm 2. Note that we include liquids (i.e., water and lava), although no houses should be placed on them. This is not a problem, however, because we later use the water and lava map (see Figure 4.3) to constrain the possible locations for buildings (see Section 4.3).



(a) Watermap.



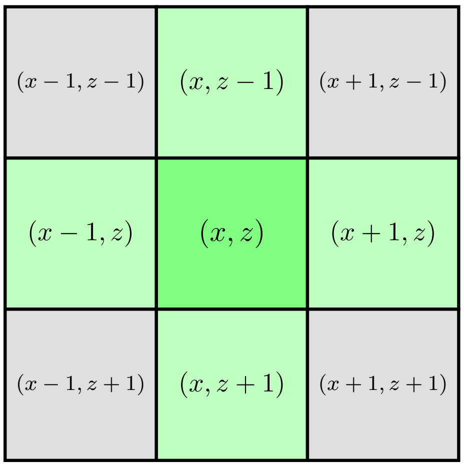
(b) Lavamap.

Figure 4.3.: The two liquidmaps of the region.

```
  input: The heightmap, watermap, and lavamap.
  output: Placement weights for the houses.
1 find connected areas;
2 find biggest connected area;
3 determine buildable area;
4 calculate flatness weights;
5 calculate the initial village center;
6 calculate house placement weights;
7 restrict buildable area;
8 initialize a global, empty list of houses;
```

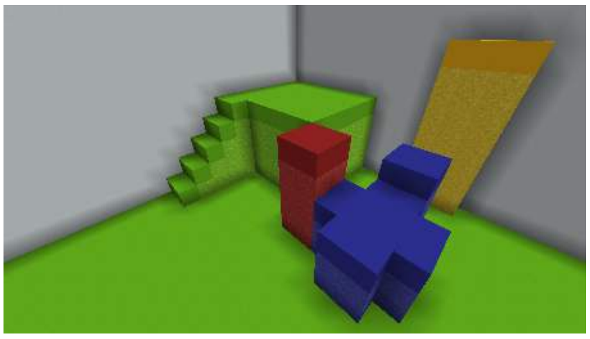
Algorithm 3: Initializing the village.

It is important to find these connected areas because we will later place all houses within the same connected area. The reason for this is that the ants that will find paths between the buildings move according to rules similar to the ones just mentioned. If two buildings were located in different connected areas, it would be impossible for the ants to travel from one to the other. A simple visualization of connected areas can be seen in Figure 4.5.



Source: Author's graphic.

Figure 4.4.: The blocks that belong to the von Neumann neighborhood of Block (x, z) are marked in light green.



Source: Author's image.

Figure 4.5.: A simple visualization of connected areas. In this scene, there are four connected areas, marked by differently colored blocks. For example, we cannot go from the green platform to the red one, because we would have to go diagonally. Likewise, we cannot jump from the red platform to the blue one because it is two blocks lower. The stairs inside the green area only go up one block per step, so the platform at its end still belongs to the green area.

Next, we simply find the largest connected area, that is, the connected area consisting of the most blocks. For this, we only count the topmost blocks, that is, the blocks that make up the heightmap. All of the following steps are performed exclusively within this area, since the others are of no interest to us. This is intuitive, since we naturally want to have as much space as possible for our village. We could not achieve this if we were to choose a small connected area. As can be seen from the example below, water and lava are included in this area. This raises a question. Can it happen that we choose an area that is the largest according to the above definition, but after excluding water and lava (which we do not want to build on) is no longer the largest? The answer to this is theoretically yes, but practically no. While such situations are possible in Minecraft worlds, they are extremely unlikely and we can therefore just ignore this issue.

In the next step, we determine the buildable area, or in other words the area where the center of houses will be placed later. A block is part of the buildable area if it meets all of the following conditions:

1. It is part of the largest connected area.
2. Let $l_H \in \mathbb{N} \setminus 2\mathbb{N}$[23]. If this block were to be the center of an $l_H \times l_H$ house[24], then:
   a) There is at least one block of space between the house and the edge of the region.
   b) There is at least one block of space between the house and other connected areas.
   c) There is at least one block of space between the house and water.
   d) There are at least three blocks of space between the house and lava.

Condition 1 comes from the fact that, as we just said, we want to have the largest possible space for our village. The idea behind condition 2 is that a house should not extend into unreachable areas and should have some additional space to them. If a house were directly adjacent to the edge of the area, for example, and the door of the house was facing the edge, a villager would have to leave the area if they wanted to exit their house. The extra distance from Lava is due to Lava's previously mentioned ability to set blocks on fire. A visualization can be seen in the example below.

Since houses should be placed on as flat a surface as possible, for each block in the buildable area, the flatness of its surrounding is calculated. This is converted into a weight, where a higher weight means a flatter area around that block. For the details, see Appendix B.

At this point, there is no village center yet, simply because there are no houses in our village. Nevertheless, we want the buildings to be placed around a "point of interest" from the beginning of the generation. To do this, we simply calculate the arithmetic mean of the buildable area, that is, the average x- and z-coordinates of all the blocks in the buildable area. Note that the coordinates of the village center do not have to be integers, since the center is only used as a heuristic for placing houses. We are not interested in the actual block in the village center.

In the next step of Algorithm 3, we calculate the weights for placing houses. Since we will later choose the placement of houses stochastically, a good place for a building should have a high weight. As indicated earlier, flatness and the Euclidean distance to the village center are taken into account to calculate the weights for placing houses. The exact definition of this weight ω_P is defined in Appendix B.

Although we could stop at this point and go to the next algorithm, we chose to introduce another mechanic. We further limit the area in which houses can be placed, but let it grow larger and larger with more iterations. The reason for this is simple. If the region where we want to generate a village is very large, despite the placement weights (B.4), it sometimes happens that houses are placed extremely far away from the center and other houses. To prevent this, we limit the buildable area to a square of side length

$$l_B := \left\lfloor \frac{n_H \cdot l_H}{2} \right\rfloor \tag{4.1}$$

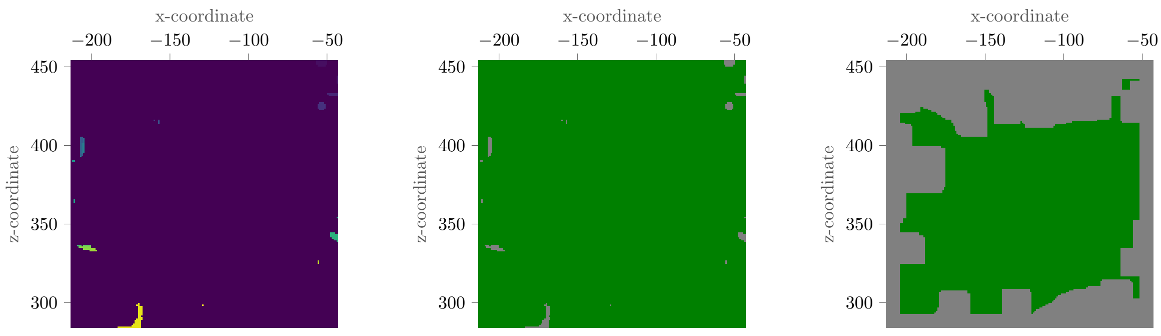
centered around the village center, where $n_H \in \mathbb{N}$ is the number of houses placed per iteration and $l_H \in \mathbb{N} \setminus 2\mathbb{N}$ is the side length of a house. In Algorithm 7, we will increase the square's side length with each iteration, thus allowing the village to occupy more area.

Last, we need a global list that contains information about all houses and villagers. Such a list does not yet exist, so we need to create it.

[23] $\mathbb{N} \setminus 2\mathbb{N}$ is the set of odd natural numbers.

[24] For simplicity, we assume that all lots have a square floor plan and that their side length is an odd number. The actual house on the lot does not have to occupy the entire lot and can have a different shape as long as it does not extend beyond the lot's boundaries. When we speak of $l_H \times l_H$ areas, we actually mean the dimension of the lot, but will continue to refer to them as houses.

Example of Algorithm 3. We continue with our example where we have $n_H = 8$ and $l_H = 15$. Figure 4.6(a) shows all the connected areas of the region, and Figure 4.6(b) shows the largest of these. Although the region is very flat, there are several smaller connected areas, so it is important that all houses are placed in the same one. Figure 4.6(c) shows the buildable area of the region. If we compare this with Figures 4.3(a), 4.3(b), and 4.6(b), we can clearly see for what reasons the area was limited. The flatness weight can be seen in Figure 4.7(a). If we compare this with Figures 4.1(b) and 4.2(b), we can clearly make out where the terrain is uneven and where it is flat. Figure 4.7(b) shows the center distance weights and implicitly also shows the village center, and Figure 4.7(c) shows the resulting placement weights. Compared to Figure 4.7(a), we can clearly see that the weight decreases the more we move away from the village center. Figure 4.7(d) shows the final weights after restricting the buildable area around the village center.



Source: Author's graphic. Source: Author's graphic. Source: Author's graphic.

(a) Connected areas. (b) Biggest connected area. (c) Buildable area.

Figure 4.6.: Determining the buildable area. Different colors in Figure 4.6(a) show different connected areas. In Figures 4.6(b) and 4.6(c), the largest connected area and the buildable area, respectively, are marked in green.

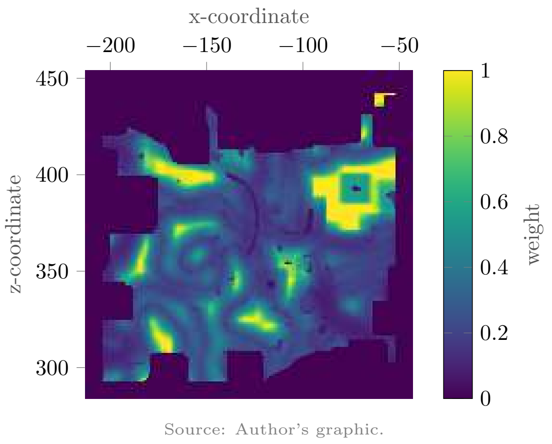
4.3. Populating the village

Now that we have made all the necessary preparations, we can place houses in the village, for which we follow Algorithm 4.

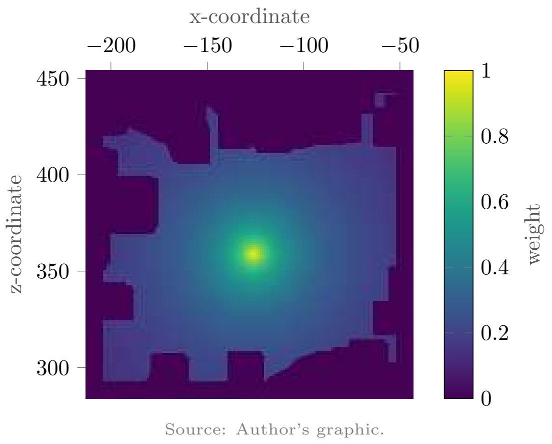
```
  input: The placement weight $\omega_P$ for each block in the buildable area.
  output: New placement weights $\omega_P$.
  updates to globals: List of houses (center block and door direction) and villagers.
1 repeat $n_H$ times
2     randomly choose a block for the center of a house according to $\omega_P$;
3     determine the house's door direction;
4     add the house and a villager living in this house to the list of houses and villagers;
5     update the placement weights;
6     if all placement weights are 0 then
7         break                                        // stop placing new houses
8     end
9 end
```

Algorithm 4: Populating the village.

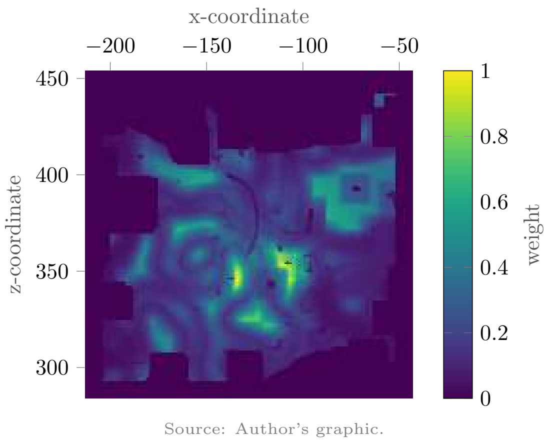
As indicated in Figure 4.7, the placement weights have been converted into a probability distribution. In the first step, which is inside a loop, we therefore do nothing but randomly



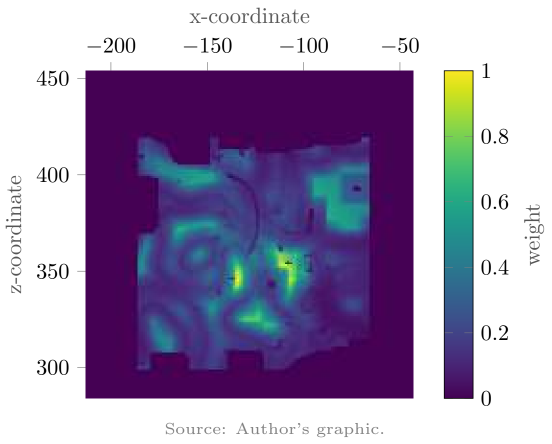
(a) Flatness weights.



(b) Center distance weights.



(c) Placement weights.



(d) Restricted placement weights.

Figure 4.7.: Calculating the placement weight for houses. In the actual implementation of the algorithm, the values are converted to a probability distribution.

select a block according to this distribution to be the center of a new house.

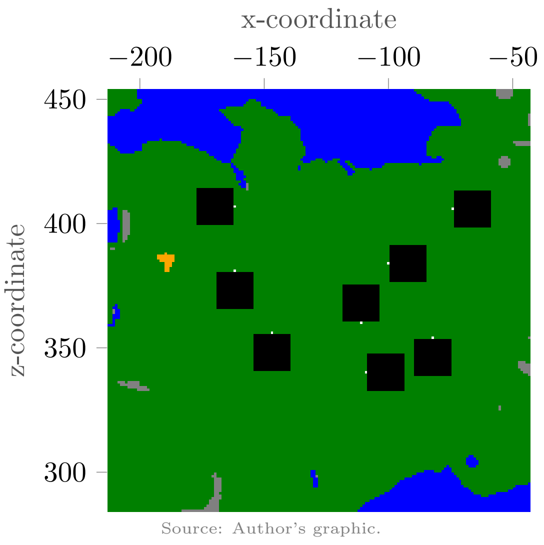
Next, we determine the direction in which the door of the house should be. For this, let $l_H \in \mathbb{N} \backslash 2\mathbb{N}$ be the side length of the house, and let us consider the four $l_H \times l_H$ squares that are directly adjacent to the four sides of the house (similar to the von Neumann neighborhood from Figure 4.4). The square that contains the most blocks that are part of the largest connected area and are not occupied by an $l_H \times l_H$ house determines the direction in which the door will be placed. The motivation for this is that the door should not be placed on a side where the villager living in that house would have very little room to move. Using this method, we expect that, for example, there is no other house just a block away from the front door[25]. After we determine the direction of the door, we add the house and the villager living there to the list of all houses and villagers. It is important that the location of the house's door is saved.

If this algorithm were to end here, it would cause problems. As can be seen in Algorithm 4, we repeat the placement of houses n_H times, where $n_H \in \mathbb{N}$ is the number of houses we want to place per growth cycle. If we were to proceed directly and place a new house, it may be

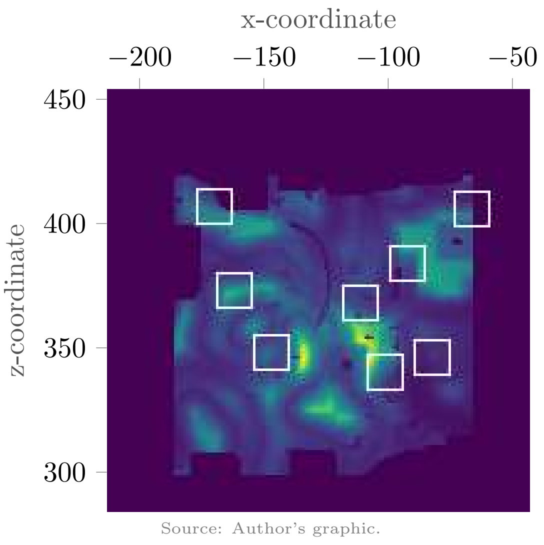
[25]Due to the placement of new houses after determining the door direction, it may happen that there is very little space in front of the door, despite our efforts. This could be solved by determining the direction of the doors of all new houses only after they are placed. However, since we will do something similar later in Algorithm 7 when the village grows, we ignore it.

placed too close to the existing house and the two buildings may overlap. To prevent this, we need to set the placement weights around the just placed house to zero and update the probability distribution. This solves the problem that houses can overlap, but can lead to all placement weights being zero at some point. If this is the case, we simply say that there is no more room in the region for new houses and break the loop for placing houses prematurely.

Example of Algorithm 4. Using the placement weights from Figure 4.7(d), $n_H = 8$ houses with side length $l_H = 15$ are now placed. The result can be seen in Figure 4.8. If we compare these two figures, we can see that flat areas are preferred and that there are more houses near the center than at the edge of the area. Moreover, we can clearly see that no house extends into other houses or inaccessible areas and that all doors are accessible[26].



(a)



(b)

Figure 4.8.: The placement of houses in the first growth iteration. Gray blocks in Figure 4.8(a) are not part of the largest connected area, but green ones are. Black blocks are houses (plots to be exact) and the single white blocks right next to them are their doors. Additionally, water and lava are marked in blue and orange, respectively. Figure 4.8(b) shows the weights from Figure 4.7(d) marked with the boundaries of the houses.

4.4. Simulating life

The algorithm explained in this section is the heart of this work and the part on which the research focuses on. As the title suggests, we now want to simulate life in the village. To be precise, the life of the villagers in the village is simulated. We are less concerned with every little detail of a villager's life, but are much more interested in how they move around the village. In fact, we will simulate the movements of the villagers and then use this information to place paths in the village. An overview of this procedure is shown in Algorithm 5.

4.4.1. Initialization

The first step, as in Algorithm 4, depends on whether the village has already grown, and if not, we need to initialize a so-called pheromone matrix. This naturally raises the question,

[26]Notice that the doors are placed one block outside the house. We decided to do this so villagers can omit the steps out of the house and start directly outside if they want to go somewhere.

```
input: The heightmap of the region.
output: None.
updates to globals: Pheromone matrix.
if first growth cycle then
    initialize a global pheromone matrix;
end
repeat numberOfSimulationCycles times
    foreach villager do
        randomly choose a house to go to;
        walk to that house;
    end
    fade old paths;                                   // evaporate pheromones
    foreach villager do
        lay the walked path;                           // secrete pheromones
    end
end
```

Algorithm 5: Simulating life.

what exactly do we mean by this matrix? The name suggests a connection with Ant Colony Optimization algorithms from Section 3.4, and this is indeed where they first come into play. Simply put, we want to simulate villagers using ants. When a villager moves through the village, it is ants that find the way for them, and the pheromones that the ants leave behind over time form a network of paths. However, to achieve this, we cannot simply use the ACO algorithms as we introduced in Section 3.4. There are some key differences between classical applications and our specific needs, of which we will briefly list the most important ones.

1. When ACO algorithms are used, the goal is to find an optimal path from one node in a graph to another. However, we want to simulate many different villagers that do not have the same starting point and destination most of the time.

2. Ants in classical ACO algorithms usually follow only very simple rules (heuristics) and do not get any preliminary information about the properties of the path they found. What this means is that an ant that is in the middle of finding a path is not influenced by the path it has found so far. However, we think that a villager will choose a different path depending on how they walked before. To be precise, we want to simulate exhaustion. A villager who has recently climbed up or down blocks prefers not to climb anymore for a short while because they are exhausted.

3. In classical ACO algorithms, the pheromones left by the ants lie on the edges between the graph's nodes. In our algorithm, however, the pheromones lie on the blocks (i.e., the nodes). This is not possible in classical ACO algorithms, since it is not important which node is traversed, but which edge is traversed into and out of the node. In our case, however, a block be visited from multiple sides, such as it is the case in the center of a path intersection. Thus, we are more interested in the fact that this block is a "busy" block than in which directions the villagers go.

With this information, we can now redefine all parts of the ACO algorithms from Section 3.4 to fit our needs, and introduce new concepts where necessary. First, let us continue with the pheromone matrix from Algorithm 5. As just explained, the pheromones lie on the blocks. Hence, the pheromone matrix does nothing but store the pheromone level for each block in the region. However, since we need to initialize this matrix in the first iteration, the question is how to do this. It is intuitive that each block should have the same value at the beginning, because initially there are no paths yet and therefore no block has been traversed more often than another. Thus, it remains only to figure out what this value is. The larger this is, the longer it takes for the paths found by the villagers to influence their pathfinding, and the smaller it is, the more the villagers are influenced by the paths found early on [8]. As with many other constants later, a good value depends on the choice of all other constants. In this thesis, we initialize all pheromone levels to one and leave this value unchanged throughout all following experiments.

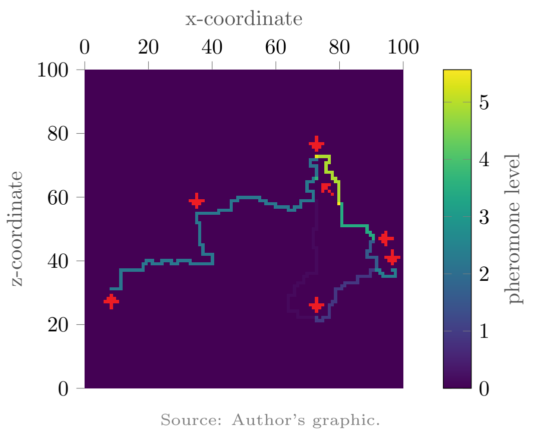
4.4.2. Walking to houses

Next, the algorithm enters a loop in which all villagers independently choose a house in the village to go to. This is done randomly and can be any house in the village (except the villager's own). The reason for this is that at this point neither villagers nor houses have functions[27], so we assume that there is no bias in the selection of houses. This also has two advantages. First, it ensures that all houses are visited about the same number of times, giving villagers a chance to connect each building to the path network. Second, it naturally allows many villagers to traverse the village center, creating a major path that can be used by many villagers.

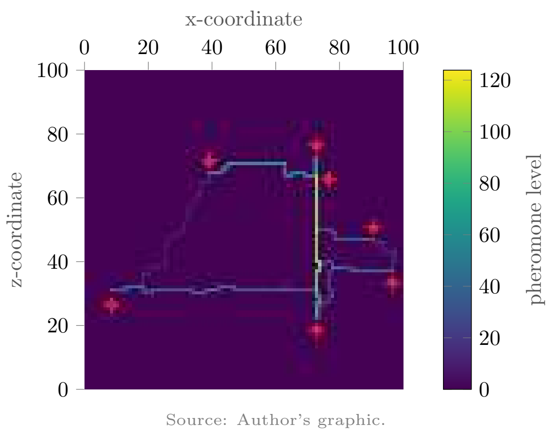
In the next step, each villager walks from their house (the block with the house's door) to the one that was selected for them in the previous step. This is where the first ACO algorithms are applied. As indicated earlier, ants will find the way for the villagers. Before defining the algorithm, we need to decide how many ants to simulate with. Intuitively, we would assign one ant per villager, since, after all, a villager cannot split up and search for multiple paths at the same time. However, this contradicts the concepts of classical ACO algorithms and actually does not yield good results in practice. This makes sense, since the goal of ACO algorithms is that of emergence, i.e., that many smaller parts in a large system work together to find a solution (see Section 3.3). With only one ant per villager, we would completely disregard this concept and hope that a single ant would randomly find a good path over time. A valid objection would be that with one ant per villager we would not have just one ant, but in fact as many ants as there are villagers. That is correct, but it still does not work sufficiently. The ants will influence each other, because they all act in the same world and see the same pheromones, but they all follow their own individual goal. A pheromone network is formed, but it is not quite what we want it to be. The problem is that in each iteration, since there are no alternative paths for comparison, the quality of a found path does not matter. That is, no matter how good a path is, as long as it is found, it always leaves a strong trace in the world[28]. If other ants were also looking for a path from the same start to the same destination, they might find a better path, which would then be more strongly marked. However, with only one ant, a path found early quickly becomes part of the final network, regardless of its quality. We have found that as little as four ants per villager are sufficient to achieve satisfactory results in a reasonable time. As for the interpretation of this approach, we can simply say that each ant simulates a day in the villager's life, and that villagers mark their paths only after these days have passed. A visual comparison of one and four ants per villager is shown in Figure 4.9 below.

These figures show the pheromone heatmaps of the same 100-by-100 region. It is to be expected that four ants per villager will converge faster than just one ant per villager, as there are simply more agents to search per iteration. To compensate for this, some parameters were adjusted accordingly. The most important of these is the number of iterations. While villagers with four ants go through 48 iterations, villagers with one ant go through 192, four times as many. Seven villagers are simulated, so seven and 28 ants, respectively. Each

[27]In fact, villagers will never have a real function. Only the houses are assigned a function later in Algorithm 8.
[28]This still happens when "bad" paths are marked very weakly, which they are in the example below.



(a) One ant per villager.



(b) Four ants per villager.

Figure 4.9.: The influence that different numbers of ants per villager have on the simulation.

small square shows the amount of pheromones on the corresponding block after 48 and 192 iterations of four villagers, respectively, with each villager starting on one of the blocks marked by the red arrows and moving to one of the others. The terrain is completely flat, which means that paths should make as few wavy lines as possible. We can clearly see that the villagers in Figure 4.9(b) have indeed formed paths that go run quite straight, but those from Figure 4.9(a) have not. Some paths make turns that unnecessarily lengthen them. Moreover, the paths from Figure 4.9(b) form more of a network than those in Figure 4.9(a) do. With only one ant per villager, no real network is formed, but instead a single path connecting all points.

We can now define how the villagers find paths. An overview for this is shown in Algorithm 6. A few new variables are introduced, which we want to explain first. The variables $B_S = (x_S, z_S)$, $B_D = (x_D, z_D)$, $B_P = (x_C, z_C)$, $B_N = (x_N, z_N)$, and $B_L = (x_L, z_L)$ are the coordinates of different blocks. B_S and B_D are the start and destination blocks, respectively, B_P is the ant's current position, B_N is the block to which the ant will go next, and B_L is the block which the ant has been to last. The function d_M calculates the Manhattan distance between two blocks according to the following formula:

$$d_M\left((x_1, z_1), (x_2, z_2)\right) := |x_1 - x_2| + |z_1 - z_2|\,. \tag{4.2}$$

Recall that $Y(x, z)$ is the y-coordinate of a block (x, z).

First, we set the number of failed pathfinding attempts to zero. Each time an ant fails to find a path, we increase this value, and abort the search if the ant has failed too many times. This is so that the algorithm does not get stuck if an ant repeatedly has trouble finding the path. Next, we set the maximum acceptable path length to four times the Manhattan distance between the start and destination blocks. The Manhattan distance is always less than or equal to the shortest possible path length between two blocks (ants move to blocks in their von Neumann neighborhood, see below). If the path traveled by the ant becomes longer than this value, we stop the search. We want the ants to deviate from the optimal path, but they should not wander around too much, and we think that no path should be longer than 4 times the shortest path length[29].

After that we enter a loop, which is repeated until one of certain conditions is met. The ant's current position is set to the starting block and this block is set to be the sequence

[29]It may happen that it is impossible to find a path shorter than four times the Manhattan distance between two houses. For example, if two houses are right next to each other but separated by a long wall or something similar, the villagers will never find a way between the two. However, such a situation is so exceptionally rare that we can ignore this problem.

```
input: The start $B_S$ and the destination $B_D$.
output: The found path (visitedBlocks).
foreach ant do
    numberOfFailedAttempts ← 0;
    maxAllowedPathLength = $4 \cdot d_M(B_S, B_D)$;
    loop
        $B_P \leftarrow B_S$;
        visitedBlocks ← $(B_P)$;      // This is a sequence, that is, an ordered
                                         set. Later, the sequence of visited
                                         blocks will be denoted by $\mathcal{B}$.
        stepsSinceLastClimb ← stepsNeededToRecover;
        while $B_P \neq B_D$ do
            $B_N$ ← choose a block to go to;
            if $B_N$ = none then
                numberOfFailedAttempts += 1;
                break
            end
            $B_L \leftarrow B_P$;
            $B_P \leftarrow B_N$;
            currentPathLength += 1;
            if currentPathLength > maxAllowedPathLength then
                numberOfFailedAttempts += 1;
                break
            end
            if $Y(B_P) \neq Y(B_L)$ then
                stepsSinceLastClimb ← 0;
            end
            else
                stepsSinceLastClimb += 1;
            end
            append $B_P$ to visitedBlocks;
        end
        if $B_P = B_D$ then
            return visitedBlocks
        end
        if numberOfFailedAttempts > maxNumberOfSearches then
            visitedBlocks ← ();      // This is the empty sequence.
            return visitedBlocks
        end
    end
end
```

Algorithm 6: A villager's pathfinding.

of blocks visited so far, meaning the sequence now contains a single element. In addition, we introduce a variable that shows how many steps (i.e., blocks) the ant has taken since it last climbed up or down a block. We initiate this value as the number of steps an ant must take to stop being exhausted. In other words, if `stepsSinceLastClimb` is less than `stepsNeededToRecover`, the ant is exhausted, which will have implications for the next step.

We enter another loop, which we execute as long as the ant's current position is not the destination block. First, we determine the block to which the ant should go next. Ants can go to a block that satisfies the following conditions:

1. It is in the von Neumann neighborhood of the ant's current position (see Figure 4.4).
2. It is not more than one block lower or higher than the ant's current position.
3. It is not inside a house.
4. It has not been visited before.

If we compare this with the rules from Section 4.2, we can see that the first two are virtually identical. This, and the fact that all houses, including their doors, are placed inside the largest connected area, now implies that an ant can never go to a block outside the largest connected area. The second to last point is also intuitive, as it prevents the ants from passing through houses. The last point is added for efficiency since it makes the ants avoid backtracking. We call the set of all blocks satisfying these three conditions for a block $B = (x, z)$ the neighborhood $\mathcal{N}_B$ of B.

Let us now define how to select the next block. Let $B_0 = (x_0, z_0)$ be the block where the ant is currently located, $\mathcal{N}_{B_0}$ be the neighborhood of B_0, and α, β, and $\gamma \in \mathbb{R}_{\geq 1}$ be positive real numbers greater or equal to one. The probability P_{B_0} of selecting a block $B \in \mathcal{N}_{B_0}$ is:

$$P_{B_0}(B) := \frac{(\tau_B)^{\alpha} (\eta_{B_0,B})^{\beta} (\theta_{B_0,B})^{\gamma}}{\sum_{\tilde{B} \in \mathcal{N}_{B_0}} (\tau_{\tilde{B}})^{\alpha} (\eta_{B_0,\tilde{B}})^{\beta} (\theta_{B_0,\tilde{B}})^{\gamma}}. \tag{4.3}$$

In the numerator, we calculate the weight of block B and then normalize it with the denominator. Notice that this formula is very similar to (3.1), but we use three heuristics instead of just two to determine the weight of a block. With the three constant exponents α, β, and γ we can influence the impact of the different heuristics, which we will examine in more detail later with a concrete example. The first is the pheromone level τ_B of block B. There is no real difference from classical ACO algorithms except that, as mentioned earlier, the pheromones are not on edges between blocks, but on the blocks themselves. To be precise, an ant tends to go to blocks that other ants have traversed in previous iterations. The more ants have traversed the block, and the more recently they did so, the more likely the ant is to choose this block.

The second heuristic is the distance weight $\eta_{B_0,B}$ of block B. This weight is higher the closer a block is to the target block. The idea behind this is that a villager never walks around completely randomly, but always tries to walk in the direction of their goal. A first naive implementation of this would be

$$\hat{\eta}_{B_0,B} := \frac{d_M (B_0, B_D) + 1}{d_M (B, B_D) + 1}, \tag{4.4}$$

where d_M is the Manhattan distance from (4.2), $B_D = (x_D, z_D)$ is the destination block, and the addition of one in the denominator is to avoid division by zero (to compensate, we also add one in the numerator). This does favor a block closer to the target, but it is not a good implementation. The problem is that this formula depends on how far B_0 is from the destination. If this distance is large, then this weight is almost one, since $d_M (B_0, B_D)$ and $d_M (B, B_D)$ always differ by exactly one block (a block in the von Neumann neighborhood of B_0 is always either one block closer to the target or one further away). On the other hand, if this distance becomes smaller, the weights move further away from one. In short, this heuristic depends on the ant's current position. However, we want to have a bias of the same magnitude everywhere. To achieve this, we normalize the weights to a fixed interval

$[H_{\min}, H_{\max}]$ around 1, where $H_{\min} \in (0,1)$ and $H_{\max} \in (1,\infty)$:

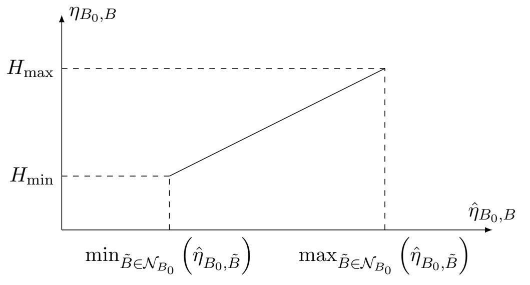
$$\eta_{B_0,B} := H_{\min} + \frac{\left(\hat{\eta}_{B_0,B} - \min_{\tilde{B}\in\mathcal{N}_{B_0}}\left(\hat{\eta}_{B_0,\tilde{B}}\right)\right)(H_{\max} - H_{\min})}{\max_{\tilde{B}\in\mathcal{N}_{B_0}}\left(\hat{\eta}_{B_0,\tilde{B}}\right) - \min_{\tilde{B}\in\mathcal{N}_{B_0}}\left(\hat{\eta}_{B_0,\tilde{B}}\right)}. \tag{4.5}$$

A graphical representation of (4.5) can be seen in Figure 4.10(a).

The last heuristic is the exhaustion weight $\theta_{B_0,B}$. As explained earlier, we want to use this to simulate the exhaustion of the ants or, in other words, the villager. Let $s_{\text{last}} \in \mathbb{N}_0$ be the number of steps since the last climb up or down a block (i.e., `stepsSinceLastClimb` from Algorithm 6) and $s_{\text{recover}} \in \mathbb{N}_0$ be the number of steps after which an ant is no longer exhausted (i.e., `stepsNeededToRecover` from Algorithm 6). The exhaustion weight $\theta_{B_0,B}$ is defined as:

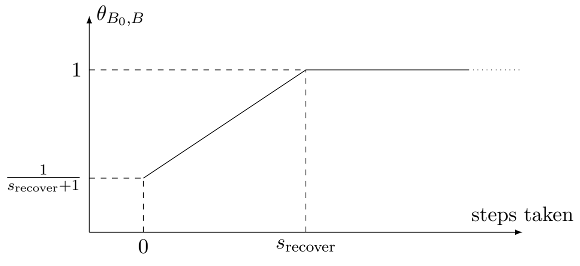
$$\theta_{B_0,B} := \begin{cases} \frac{\min(s_{\text{last}}, s_{\text{recover}})+1}{s_{\text{recover}}+1}, & \text{if } Y(B_0) \neq Y(B), \\ 1, & \text{otherwise.} \end{cases} \tag{4.6}$$

In other words, if an ant has just recently climbed, we reduce the weight of all the blocks for which the ant would have to climb again. Once it has fully recovered, there is no such penalty. The graph of this function's first case is shown in Figure 4.10(b).



Source: Author's graphic.

(a) Distance weight normalization.



Source: Author's graphic.

(b) Exhaustion weight.

Figure 4.10.: Visualizations of the weights for block selection.

Now that the ant has selected a block, we can proceed with Algorithm 6. Sometimes it can happen that the neighborhood of a block is empty. This is either the case when the ant has run into a dead end, or when it has already visited all the neighboring blocks. In this case, since it is stuck, the ant aborts its search. We also increase the number of failed pathfinding attempts by one. Once the ant has found a block, it travels to this block, while also remembering where it was before. If the ant has exceeded the previously set maximum path length by doing so, it aborts and increases the number of failed pathfinding attempts by one. If this is not the case, the ant checks whether it had to go up or down a block. If it had to do so, its steps since the last climb are reset to zero, or in other words, it is exhausted. Otherwise, we increase the steps by one, since the ant was able to recover a bit more. Last, it appends its current position to the sequence of all blocks visited so far.

The ant repeats all these steps until it either arrives at its destination or has to stop its search prematurely. Once it has reached its destination, it saves all the blocks it has visited on its path and then steps out of the search loop, allowing the next ant to search for a path. If it has not reached the destination and instead has exceeded the maximum number of attempts, it also aborts the search, but in this case it forgets all the blocks it visited in its last attempt. In other words, the ant has not found a path. If neither of these situations is the case, it starts its search from the beginning.

4.4.3. Fading old paths and laying new ones

Once all villagers have finished Algorithm 6, we simulate the fading of old paths. This can be interpreted as paths weakening because they weather, grass grows over them, and so on. Accordingly, this is identical to the pheromone evaporation of classical ACO algorithms from Section 3.4 and, with the exception of the indexing, the formula is identical to (3.2):

$$\tau_B \leftarrow (1-\rho)\,\tau_B, \tag{4.7}$$

for all blocks B of the region, where $\rho \in (0,1]$ is the pheromone evaporation rate.

It now only remains to secrete pheromones on the newly found paths. To do this, we need to determine how large the amount of pheromones should be depending on the path found. Classical ACO algorithms usually use the path length for this, since they are an optimization procedure that wants to find the shortest path. We, on the other hand, are not interested in the shortest path, but in the one that looks most natural. We have explained the criteria for this in detail in Section 2.3.1. In short, a path should not rise and fall too much, but should also not be unnecessarily long. To penalize paths that are too long, we can compare a path similar to (4.4) to the shortest path between start and destination when calculating distance weights. For penalizing paths with a lot of up and down, we can look at how much the path has risen and fallen locally, or in other words its local slope. Unlike the classical definition of a slope, we mean how many times an ant has climbed within a fixed distance. Let $r \in \mathbb{N}$ be then run, that is, the number of blocks over which we want to compute the slope, let $\mathcal{B}$ be the sequence of blocks the ant has visited along its path, and let $\mathcal{B}_r := (B_k)_{k\in\{1,\ldots,r\}} \subset \mathcal{B}$ be a sequence of r blocks that the ant has visited in succession. The slope S_r is then defined as

$$S_r(\mathcal{B}_r) := \frac{\sum_{k=1}^{r-1} |Y(B_k) - Y(B_{k+1})|}{r-1}, \tag{4.8}$$

where $Y(B_k)$ is the y-coordinate for Block B_k for all $k \in \{1,\ldots,r\}$ as before[30]. If all blocks are at the same height, the slope is zero. However, if two consecutive blocks are always at different heights, then the slope is one. In between, it increases linearly. With this, we can now define how many pheromones are to be secreted on a path. Let $\mathcal{B}$ and r be as just defined, let $B_S, B_D \in \mathcal{B}$ be the start and destination blocks, respectively, and let $\phi, \chi \in \mathbb{R}_{\geq 1}$ be positive real numbers greater or equal to one. The evaluation Δ of path $\mathcal{B}$ is defined as

$$\Delta(\mathcal{B}) := \left(\frac{d_M(B_S, B_D)}{|\mathcal{B}|-1}\right)^{\phi} \left(1 - \frac{r \max\left(S_r\left(\mathcal{B}_r\right)\right)}{1+r}\right)^{\chi}, \tag{4.9}$$

where we can use ϕ and χ to adjust the influence of the two individual parts. The first factor is closer to one the closer the path length (the number of blocks on the path minus one) is to the Manhattan distance between start and destination, and tends towards zero otherwise. The second factor is closer to one the closer the maximum local slope is to zero, and tends to $1/(1+r)$ otherwise. Thus, overall, paths that are not unnecessarily long and bumpy are evaluated to be better, and accordingly, more pheromones are secreted on them. Now the ant follows the formula

$$\tau_B \leftarrow \begin{cases} \tau_B + \Delta(\mathcal{B}), & \text{if } B \in \mathcal{B}, \\ \tau_B, & \text{otherwise.} \end{cases} \tag{4.10}$$

to emit the number of pheromones calculated in (4.9) on all blocks of its path. This is similar to a combination of (3.3) and (3.4).

If we take a look back at Algorithm 5, we see that we have reached the end. We now repeat all the steps introduced in this section `numberOfSimulationCycles` times. In every

[30] If $\mathcal{B}$ has fewer than r elements, we calculate the slope over all blocks of the path instead and adjust r accordingly.

iteration, each villager searches for a path to a random house, influenced by pheromone paths that already exist. Afterwards, all paths fade out a bit, and all villagers lay their newly found paths.

4.4.4. Influence of the constants

In Algorithms 5 and 6, many constants are introduced which must be specified for an implementation. A good selection is important for good results, and if different results are to be obtained, the parameters must be adjusted accordingly. We will therefore explain the influence of all constants in more detail in this section. Where it makes sense, we will also compare different values using concrete examples. What we cannot do, however, is name the uniquely best values for each constant. The problem is that there are too many variables, all of which influence each other. They are also dependent on variables introduced earlier, such as the number of ants per villager. A fixed value of one constant may give good results with certain values of all the other constants, but must be changed if we change the other constants to retain these results. Instead, at the end of this section, we will continue the example of the previous sections and state exactly which constants we used to get these results.

This is not a difficulty only we had to face. Although Marco Dorigo, who invented ACO algorithms, has far fewer variables in his variants of the algorithms, he too was unable to assign a clear best value to these variables. Instead, he says that he experimented with many different values, and then cites which ones he got the best results with [8]. We will therefore do the same.

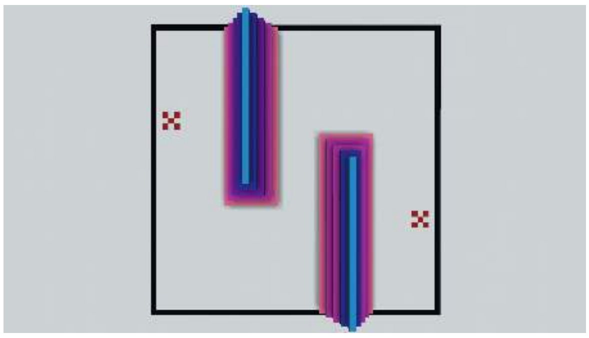
Let's start with `numberOfSimulationCycles`. If we stop too early, the villagers have not had enough time to form a network of paths. Their paths have not converged yet and there is still too much noise. Stopping too late is not really a problem, but we still want to avoid it. At some point, a stable network will have formed and villagers will rarely deviate from it. If we can estimate this point, we can stop earlier and thus save time. This is especially important for the GDMC competition. Similarly, `maxNumberOfSearches` also influences the runtime of the algorithm. Due to the precautions taken, it is (almost) always possible for an ant to find its path. There is (almost) always a way between two houses that does not exceed the length limit set by `maxAllowedPathLength`. However, it is sometimes difficult to find, for example when a rather large detour has to be taken. Instead of waiting for the ant for a long time, it makes sense to stop the search. This may result in no paths being formed to houses in hard-to-reach areas though. If time does not matter, this value should be large or entirely omitted in the implementation.

Let us now consider (4.3), (4.5), and (4.6). Since the path found depends not only on these formulas, but also strongly on (4.9), we must assume that the same number of pheromones is always released for investigating the effects of α, β, and γ. In other words, we set $\phi = \chi = 0$[31]. Let us start with the simplest value, α. There is not much to explain here. The higher α is, the more the ants follow the already existing pheromone paths. A value that is too low will cause the paths to not converge, as the ants will look for new paths much more often instead of following already known ones. A value that is too high will cause all ants to follow only the paths found in the first iteration. A good value for α depends strongly on how many pheromones are secreted later. If this number increases, but we want to get the same results, α must be decreased to counteract the effect, and analogously the other way around.

The value of β is directly related to the values of $H_{\min}$ and $H_{\max}$. We can therefore assume that $H_{\min}$ and $H_{\max}$ are fixed. It makes sense to choose $H_{\max} = 2 - H_{\min}$ to get a uniformly distributed weight for all blocks in the neighborhood of an ant's block. The smaller $H_{\min}$

[31]Notice that ϕ and χ cannot be zero by definition. We choose to set them to zero only once to examine α, β, and γ since with $\phi = \chi = 0$ in (4.9), the pheromones released on each block of a path always equal one.

and larger $H_{\max}$ is, the more often villagers go towards their destination. Furthermore, it makes sense to examine the value of γ at the same time, since the path also depends on the relationship between β and γ. The effect of γ, in turn, is directly influenced by s_{recover}, so we can also fix it. The larger s_{recover} is, the flatter the paths are that the villagers find. Let us examine all of these variables using a concrete example. For this purpose, we employ the world shown in Figure 4.11. A villager simulated by four ants walks from the left red cross to the right one. The colored blocks are obstacles which become one layer higher with each additional color. The black frame shows the borders (not included) of this 48-by-48 region.



Source: Author's image.

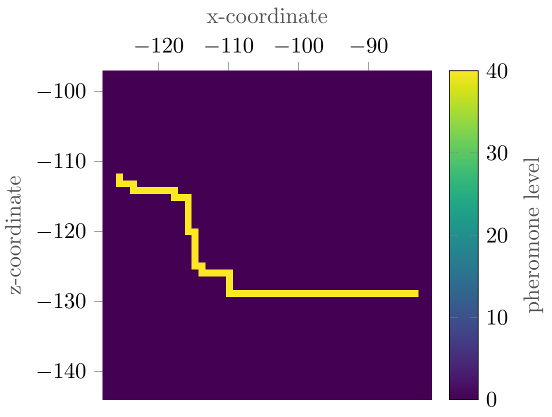
Figure 4.11.: The world used for the experiments.

When β is very large, the villager ignores the terrain and takes the shortest path to the destination. We can see this well in Figure 4.12(a). In fact, this path is exactly as long as the Manhattan distance between start and destination. If we compare with Figure 4.11, we can clearly see that the villager simply walks over the obstacles instead of around them. On the other hand, this is different when γ is very large. The villager avoids climbing up and down blocks and pays less attention to how long their path is. This is reflected in the wavy path chosen by the villager in Figure 4.12(b). The path is clearly longer, but it bypasses the obstacles completely. When we want to generate villages later, it is important to find a balance between these two constants.

Next, we consider the constants from (4.7), (4.8), and (4.9). The effect of ρ is very simple. If its value is too high, paths fade too quickly and do not converge because villagers are not enough biased to paths already found. Too low of a value, on the other hand, causes old and infrequently-used paths to persist too long, creating a lot of noise. Next, the smaller r is, the more strict we are with uneven paths. With larger r, we do not care if the path is very uneven in a small area as long as it is fairly flat on average. It makes sense to give r the same value as s_{recover}, since both determine the run of a slope and have the same goal.

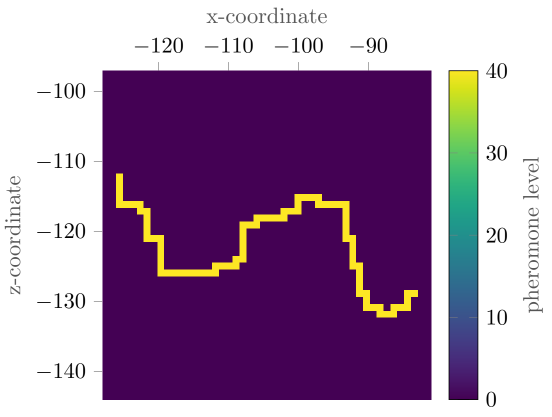
The two constants ϕ and χ are, like β and γ, the constants whose value has the greatest influence, which is why we want to take a closer look at them in a concrete example. In order to examine these them more closely, we need to set $\beta = \gamma = 0$[32], analogous to earlier, so that this time the villager is influenced only by pheromones and no other heuristics. In fact, ϕ and χ each have the same function as β and γ, just at different steps in the algorithm. The value of β affects how far the path length deviates from the Manhattan distance between start and destination. The value of ϕ, on the other hand, determines how much a path should be penalized the further its length deviates from the Manhattan distance between start and destination. In fact, we can see from Figure 4.13(a) that the villager has found a very short path, completely disregarding the obstacles. This relationship is similar for the other two constants. The value of γ influences how often a path goes up and down, and

[32]Similar to earlier, the definition of these variables does not actually allow for this. Setting them zero is done only once, because this way the distance weight and flatness weight of each block has the same value of one. In other words, these two weights are ignored.



Source: Author's graphic.

(a) Finding a path that is as short as possible: $\beta = 10$, $\gamma = 1$.



Source: Author's graphic.

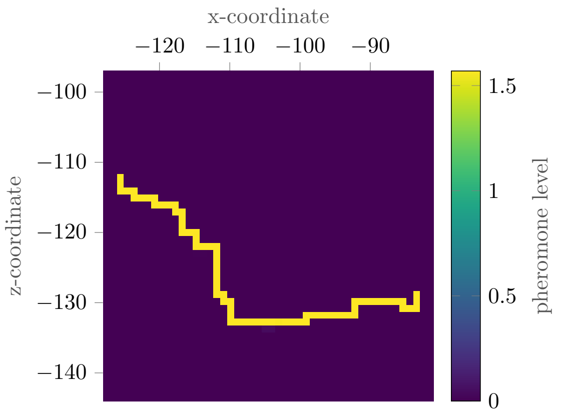
(b) Finding a path that is as flat as possible: $\beta = 1$, $\gamma = 10$.

Figure 4.12.: A demonstration of the influence of β and γ on pathfinding. The values of all other variables are the same for both examples: $\alpha = 3$, $H_{\text{min}} = 0.8$, $H_{\text{max}} = 2 - H_{\text{min}} = 1.2$, $s_{\text{recover}} = 4$, $\rho = 0.1$, $r = s_{\text{recover}} = 4$, $\phi = 0$, and $\chi = 0$. Notice that we set ϕ and χ to zero, as aforementioned. This means that each path is evaluated equally and the villager prioritizes paths based on how many times they have walked them.

the value of χ determines how strongly a path should be penalized the more it goes up and down. We would expect to get a similar result to Figure 4.12(b). However, this is not the case, as we can see in Figure 4.13(b). The reason for this is that the villager does not follow any heuristics other than pheromones during their path search. In order for them to find a snake-like path like before, they would have to find one by pure chance. Instead, the villager converges to the first path it finds that is sufficiently flat.

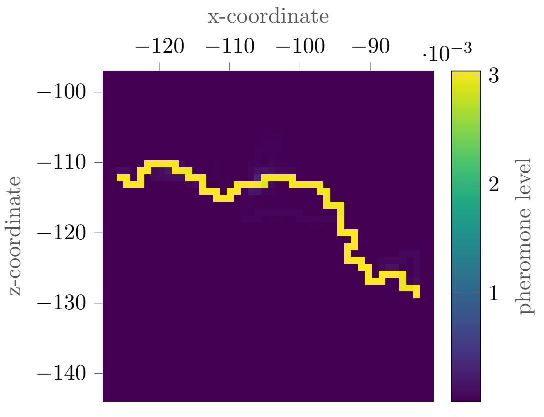
At this point, a question arises that we want to answer. Since we are effectively doing the same thing twice with the heuristics and the path evaluation, would it not be enough to do just one of them? Although the examples just shown might suggest so, the answer to this is no. The experiments were all conducted with only one villager walking to the same destination over and over again. This is a very simplified version of what we actually want to do, which is generate an entire village. If we were to use only heuristics, we cannot guarantee that the paths we find would look natural, because we do not evaluate them at all. While a villager is less likely to find an unnatural path, it is most definitely not impossible. If now the same number of pheromones are secreted on such a path as on a natural-looking one, it easily happens that the villagers converge to the unnatural path. On the other hand, if we would completely omit the heuristics, the villagers would have extreme difficulties in finding a path at all. They would wander around completely randomly, especially in the first iteration, and hope that they arrive at their destination by chance.

Example of Algorithm 5. We continue with the example from the previous section. Recall that we have placed eight houses of dimension 15-by-15 and that there lives a villager simulated by four ants in each of them. We simulate pathfinding 48 times and set the variables introduced in this section as follows: $\alpha = 3$, $\beta = 3$, $\gamma = 2$, $H_{\text{min}} = 0.8$, $H_{\text{max}} = 2 - H_{\text{min}} = 1.2$, $s_{\text{recover}} = 4$, $\rho = 0.1$, $r = s_{\text{recover}} = 4$, $\phi = 1$, and $\chi = 2$. Figure 4.14 shows the generated path network after one, 16, 32, and 48 iterations. We can clearly see how all of the villagers start out following their own path, but over time are influenced by the paths of the other villagers. After only 16 iterations, they have already formed a rough path network, but some still deviate from it. With each iteration, however, this becomes stronger, and



Source: Author's graphic.

(a) Finding a path that is as short as possible: $\phi = 10$, $\chi = 1$.



Source: Author's graphic.

(b) Finding a path that is as flat as possible: $\phi = 1$, $\chi = 10$.

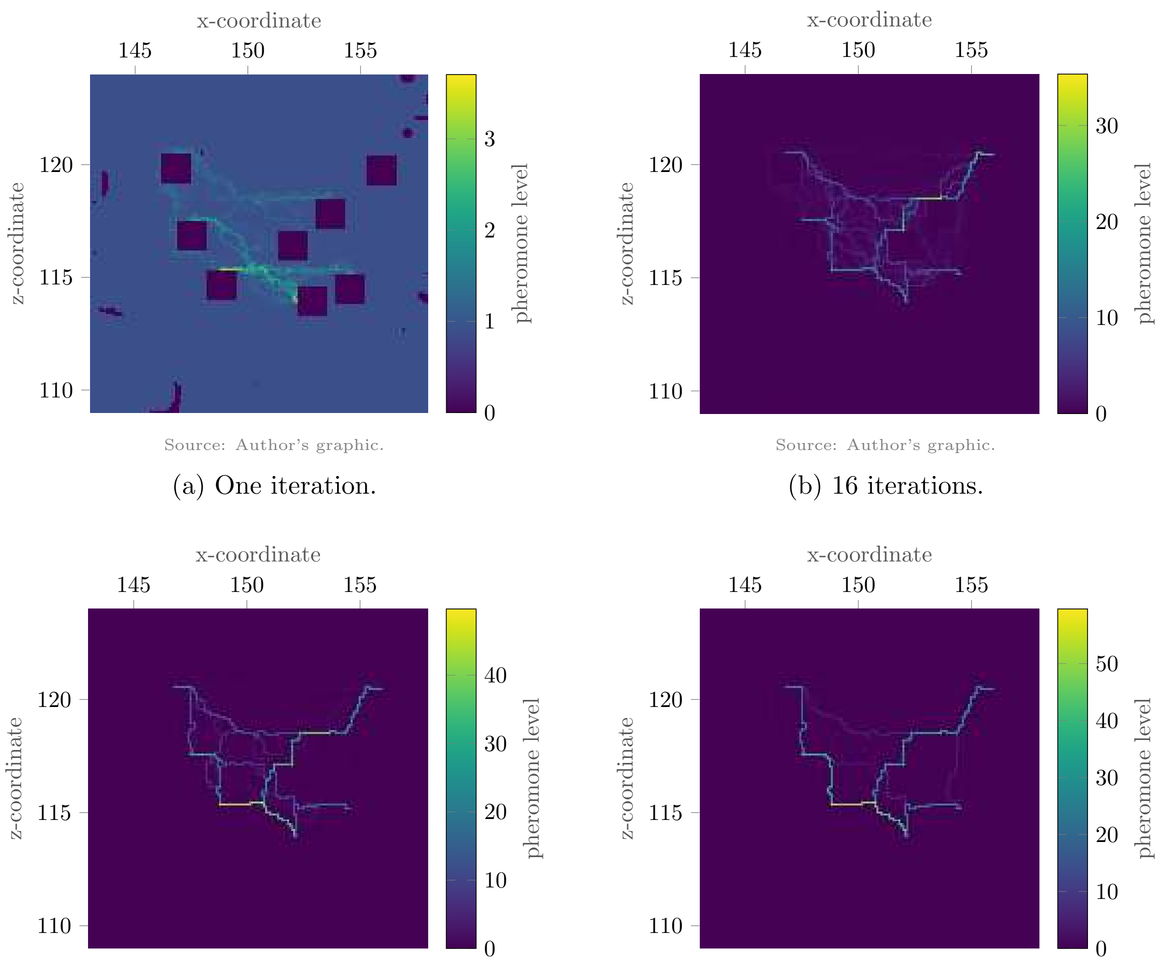
Figure 4.13.: A demonstration of the influence of ϕ and χ on pathfinding. The values of all other variables are the same for both examples: $\alpha = 3$, $\beta = 0$, $\gamma = 0$, $H_{\text{min}} = 0.8$, $H_{\text{max}} = 2 - H_{\text{min}} = 1.2$, $s_{\text{recover}} = 4$, $\rho = 0.1$, and $r = s_{\text{recover}} = 4$. Notice that we set β and γ to zero, as aforementioned. This means that the villager chooses their path based solely on the pheromone level of the blocks, and otherwise walks around rather randomly.

at the end of the simulation they have formed a strong network. This is also reflected in the pheromone levels. At the end, the maximum is more than 16 times larger than at the beginning, indicating that the found paths are traversed very often.

4.5. Growing the village

Now that we have found a path network, we could stop and jump to the next step, which is to build the village. However, if we take a look at our example, we can see that the village is quite small and there is a lot of unused space in the region. The simple solution to this would be to just increase the number of houses that are placed. However, we find that this contradicts the concept of a natural village. A village in the real world does not start out as a large collection of houses, but rather as a small one that grows larger over time. This is exactly what we want to simulate, in other words we want the village to grow. The overview for this is shown in Algorithm 7. This algorithm is basically the entire while loop from Algorithm 1, except that we do not initialize the village but instead update it. Note that instead of a time limit we could also specify exactly how often the village should grow. As we mentioned earlier, a time limit makes sense in the context of GDMC competition, but it can be easily changed. In fact, for our example, we will do just that later.

First, we update the village center. Since we are no longer in the first iteration, there are houses in the village that we can use for this. As indicated in previous sections, we simply define the center as the average center of all houses. We then expand the boundaries of the village. In Algorithm 3, we had defined the buildable area to be a square around the village center, which has a side length as defined in (4.1). If we now want to place new houses, it makes sense to extend this boundary a bit since we naturally need more space for new houses. For this, we proceed exactly the same as in Algorithm 3, but of course we use the new village center and use the following side length for the square. For this, let $i \in \mathbb{N}$ be the i-th growth iteration we are currently in, and let $n_H \in \mathbb{N}$ and $l_H \in \mathbb{N}$ be the number of houses per iteration and the side length of those houses, respectively, as before. The new



Source: Author's graphic.

(a) One iteration.

Source: Author's graphic.

(b) 16 iterations.

Source: Author's graphic.

(c) 32 iterations.

Source: Author's graphic.

(d) 48 iterations.

Figure 4.14.: Shown is the path network found at various times during the simulation. Notice that in the first iteration the position of the houses are visible. This is because in the implementation we set the pheromone level of the houses' blocks to zero. The flatness of the paths and the road hierarchy is discussed in the final result in Section 4.6.

side length of the buildable area is now defined as:

$$l_{B,i} := \left\lfloor \frac{n_H \cdot l_H}{2} \right\rfloor + i \cdot l_H. \tag{4.11}$$

Note that this may cause houses from previous iterations to end up outside the buildable area. This is not a problem, however, because the villagers have no information about the buildable area. They just move within the largest connected area.

Next, we also need to update the placement weights for new houses, since the village center has most likely moved. For this, we can follow (B.4) and simply need to change the coordinates in (B.3a). The flatness weights remain unchanged.

We then update the directions of the doors of the already existing houses. The motivation for this is that a house might be right next to a strong path, but turned away from it. The villager living in this house or any villager going to this house must therefore completely walk around the house each time to get to the bigger path. It would make more sense if the door was on the side facing the bigger path. As in Algorithm 4, we consider the four $l_H \times l_H$

```
input: The time timeElapsed since starting the algorithm.
output: None.
1 while $timeElapsed < timeLimit$ do
2     update the village center;
3     grow the borders;
4     update the house placement weights;
5     update the door directions;
6     update the pheromone levels;
7     populate the village;
8     simulate life;
9 end
```

Algorithm 7: Growing the village.

squares directly adjacent to a house for this. This time, however, we are not interested in how many blocks of these squares are within the biggest connected area. Rather, we want to know what the overall pheromone level of all the blocks within the square is. The square with the highest pheromone level contains the most developed paths, and we want the door on that side. A valid objection is that houses in villages in real life cannot just rotate, and we cannot oppose this objection. Unfortunately, it sometimes happens that the houses (especially the ones of the first iteration) point in directions that simply do not look natural. It therefore makes sense to rotate the houses, even if this slightly contradicts the natural development of the village.

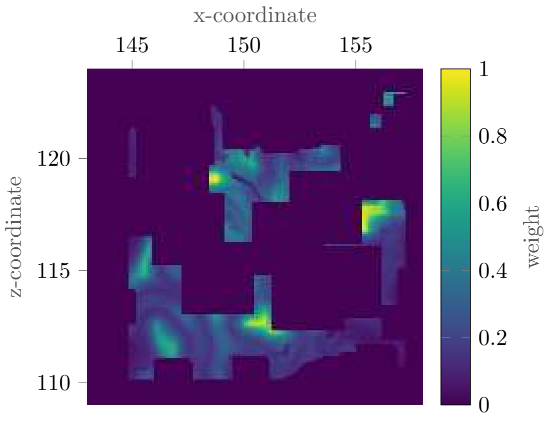
Before we place new houses and simulate life in the village again, we have one last thing to do. We have reached our goal in the first iteration in that we found a strong path network that the villagers use to move around the village. This means that the pheromone level on the found paths is very high. However, this also means that the pheromone level away from the paths is extremely small, as it decreases in each simulation cycle because of path fading (4.7). We can see this well in Figure 4.14. After 48 iterations, the highest pheromone level is almost 60, while the lowest is only about 0.006. This is great if the villagers only want to go to houses already incorporated in this network. However, if they want to go to houses that are not yet connected to the network, this is extremely difficult. The pheromone level away from the network is so low that it is extremely unlikely that they will ever select such a block. In other words, they are stuck on the existing network. Since we do not want to completely lose the paths we have found so far, we weaken them and at the same time strengthen blocks outside of them. For this, we set the lowest pheromone level to one, the highest to four, and adjust all values in between accordingly, similar to (4.5). Let B be a block in the region $\mathcal{R}$ of the world. We set:

$$\tau_B \leftarrow 1 + \frac{\overbrace{3}^{4-1}\left(\tau_B - \min_{\tilde{B}\in\mathcal{R}}\left(\tau_{\tilde{B}}\right)\right)}{\max_{\tilde{B}\in\mathcal{R}}\left(\tau_{\tilde{B}}\right) - \min_{\tilde{B}\in\mathcal{R}}\left(\tau_{\tilde{B}}\right)}. \tag{4.12}$$

The lower limit is simply the initial pheromone level. We chose the upper bound to have four times more pheromones on the strongest block than on the weakest. This way the already existing path network is not weakened too much, but at the same time not too dominant.

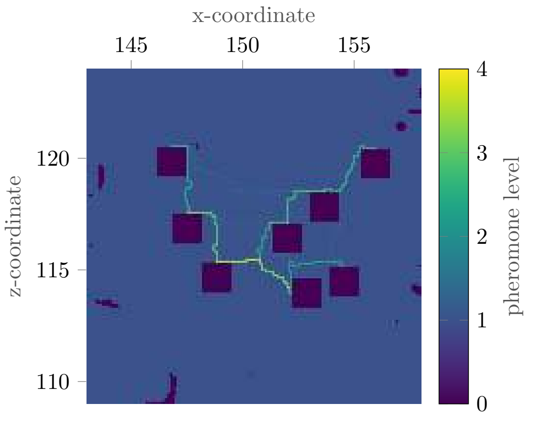
We can now place houses again and proceed as in Algorithm 4. The only difference is determining the direction of the door. As we did when updating the door directions just now, we look at the pheromone levels, not the number of blocks within the largest connected area. Note that it may happen that houses are placed on paths of the previously found path network. Afterwards, we simulate life in the village again, for which we follow exactly the same steps as in Algorithm 5.

Example of Algorithm 7. We let the village from our example grow exactly once. That is, we use a fixed limit on how many times the village can grow instead of a time limit. The new placement weights are shown in Figure 4.15(a). You can clearly see where houses were previously placed, since new houses must not overlap with these. The new pheromone levels can be seen in Figure 4.15(b). This differs from Figure 4.14(d) only in that the pheromone levels are within a different range and that the old houses can be seen more clearly (because the pheromone level of the blocks of the houses is zero). The new and old houses can be seen in Figure 4.15(c), and if we compare this to Figure 4.8, we can also see that the direction of the doors of some houses have been changed. Finally, in Figure 4.15(d) we see the resulting path network after simulating life in the village a second time (and normalizing the pheromones for better visibility).



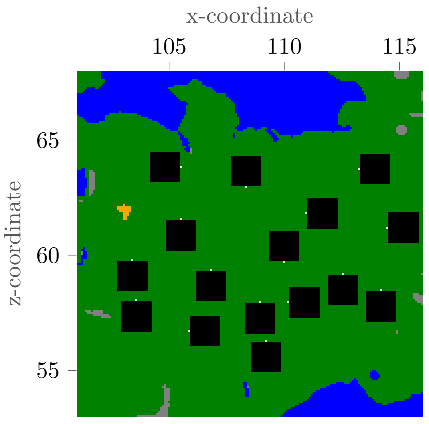
Source: Author's graphic.

(a) Placement weights.



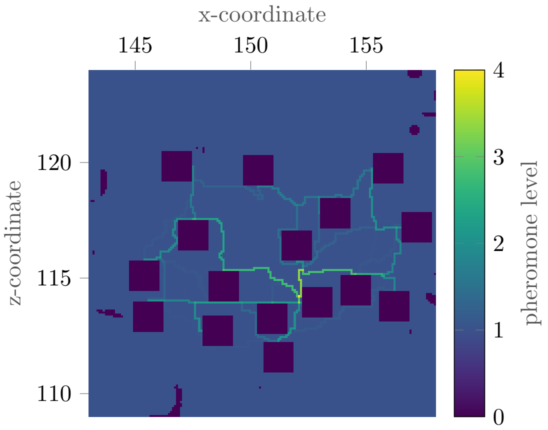
Source: Author's graphic.

(b) Pheromone update.



Source: Author's graphic.

(c) New houses.



Source: Author's graphic.

(d) New path network.

Figure 4.15.: Growing the village.

4.6. Building the village

The only thing left now is to actually build the village, that is, place it in the Minecraft world. As already mentioned, the placement of houses will follow simple rules. To be precise, houses near the village center will have different functions (hospital, etc.) than houses on the outskirts of the village (farm, etc.). Paths are placed where there are enough pheromones, and more pheromones mean that the path is more developed. An overview is shown in Algorithm 8.

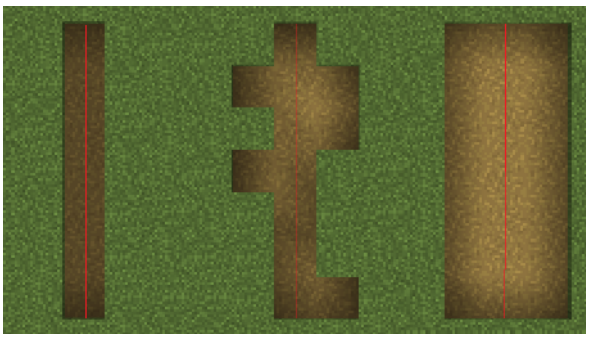
```
input: Houses, the village center, and the pheromone level of each block.
output: None.
update the pheromone levels;
foreach block B in the region $\mathcal{R}$ do
    if $\tau_B < 1.2$ then
        do nothing;
    else if $\tau_B < 2.0$ then
        place small path;
    else if $\tau_B < 3.0$ then
        place medium path;
    else
        place big path;
    end
end
update the village center;
foreach house do
    if the house is close to the center then
        assign a special function to the house;
    else if the house is far away from the center then
        make the house a farm;
    else
        make the house a normal residental house;
    end
    build the house;
end
```

Algorithm 8: Building the village.

The first step is to place all the paths in the village. Before doing this, however, we have to solve a similar problem as with the growth of the village. We cannot predict in which order of magnitude the pheromone levels are. To be able to distinguish much used paths from little used ones, we normalize all pheromones exactly as before. For this, we use (4.12), as in Algorithm 7, to normalize all pheromones to the interval $[1, 4]$. Now, depending on how many pheromones are on a block, we place a different type of path[33]. If the pheromone level is less than 1.2, we place no path. If it is on the interval $[1.2, 2)$, it means that the villagers have generated a path on that block, but it is little used. We therefore place a path block at that location but do not do anything else. However, as soon as the level gets above a value of 2, we place more than a single block. Let us first consider the case when the pheromone level is on the interval $[3, 4]$. This is the interval with the largest values, and accordingly it means that major paths have blocks with pheromone levels in this range. Instead of placing a path block only at the block in question, we place path blocks at all blocks in the 3-by-3 square around it as well. This makes the path much wider and gives the impression that it is used much more often and plays a more important role in the village. When the pheromone level is on the interval $[2, 3)$, we do the same, but do not always place a path block in the 3-by-3 square. Instead, a path block is placed at each block within that square with a 25% probability (the middle block is still always a path block). This makes it look like the path

[33]If different bounds are used for normalization in (4.12), the upper bounds of the if statements in Algorithm 8 must be adjusted accordingly.

is in a transition phase between the other two paths. Either the number of traverses across it increases and it transforms from a small path to a large path, or the number of traverses decreases and it goes from a large path to a small one. The path is still continuous, but it is sometimes wider and sometimes narrower. This makes perfect sense in Minecraft because path blocks are nothing more than trampled grass, that is, grass that has been stepped on a lot. The more villagers that walk across a path, the more the grass on and around it gets trampled, and if all of a sudden fewer villagers use a path, the grass slowly grows back over it. The three different paths are visualized in Figure 4.16.



Source: Author's image.

Figure 4.16.: The different types of paths. From left to right it is a small, medium, and big path. The blocks that actually had enough pheromones on them are under the red line.

Once we have built all the paths in the village, only the houses remain. This is the first place in the algorithm where each building is explicitly assigned a function. At first glance, this contradicts a village in the real world since houses always have a function from the beginning. This is true, but houses can easily change their function. Perhaps a blacksmith buys an old house and repurposes it into his forge, or demolishes it completely and builds a new building from scratch. Therefore, it makes no difference to assign a function to the houses at the beginning and possibly change it later, or to do this at the very end.

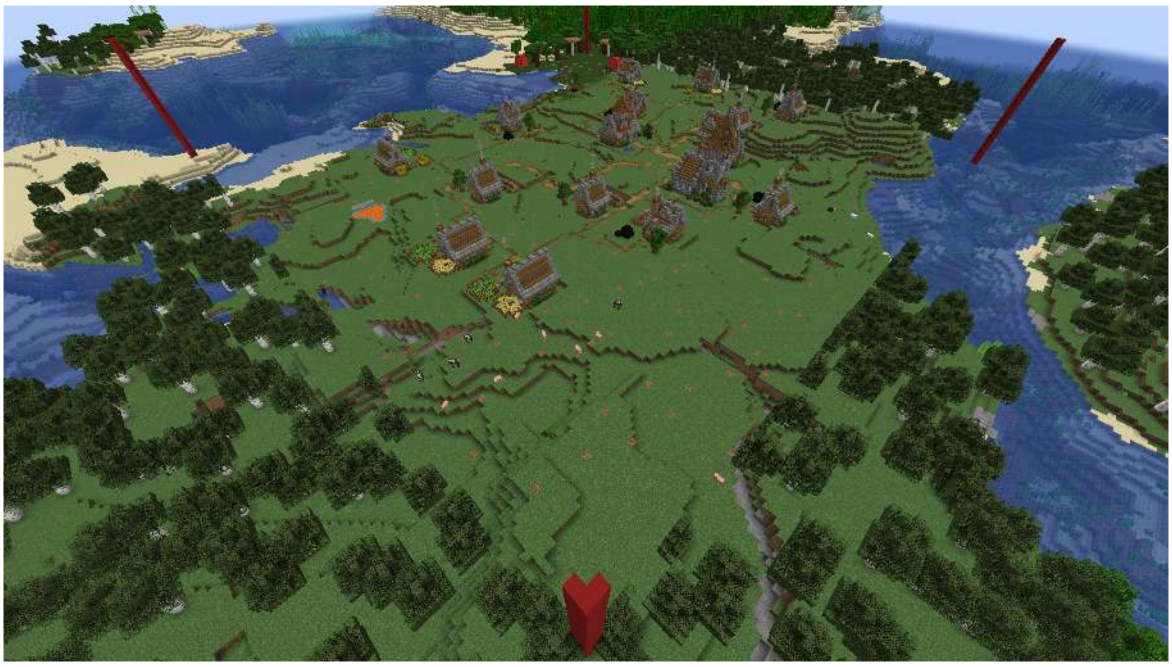
To give a house a function, we consider where it is located within the village. To do this, we first update the village center, since we have not yet done this after the last growth cycle. In our implementation there are five types of houses: a hospital, a tavern, a church, a residential house and a farm. The building closest to the village center becomes the hospital. This makes sense, since a hospital should be located where it can be reached as quickly as possible by as many people as possible. The buildings that are second and third closest to the center become the tavern and the church, respectively. Again, this makes sense, since a tavern is built where there is a lot of traffic, and villages are often built around churches. The houses that are far out become farms, and all the houses in between become normal homes.

Example of Algorithm 8. We are now ready to generate a complete village. From the previous examples, we have already gathered all the necessary information and can now place paths and houses using the algorithm presented in this section. The algorithm took a total of about two minutes to complete all steps, starting with Algorithm 2 and ending with Algorithm 8[34]. Figure 4.17 shows an overview of the entire village[35]. This screenshot is taken from the same position as Figures 4.1(a) and 4.1(b) and shows the entire region. We count 16 houses, as expected (one house hides slightly behind the hospital and two behind the church). Where the hospital, the church, and the tavern (behind the church) are, we

[34]The experiment was performed on a computer with an Intel Core i9 10900X processor (19.25M cache, 3.70 GHz) and 256 gigabytes of DDR4 memory.

[35]The design of the buildings was strongly inspired by Mechitect: `https://www.youtube.com/@Mechitect/`.

note that more houses are closer together than outside of that area. No house is placed on water, and all buildings keep enough distance from the lava.



Source: Author's image.

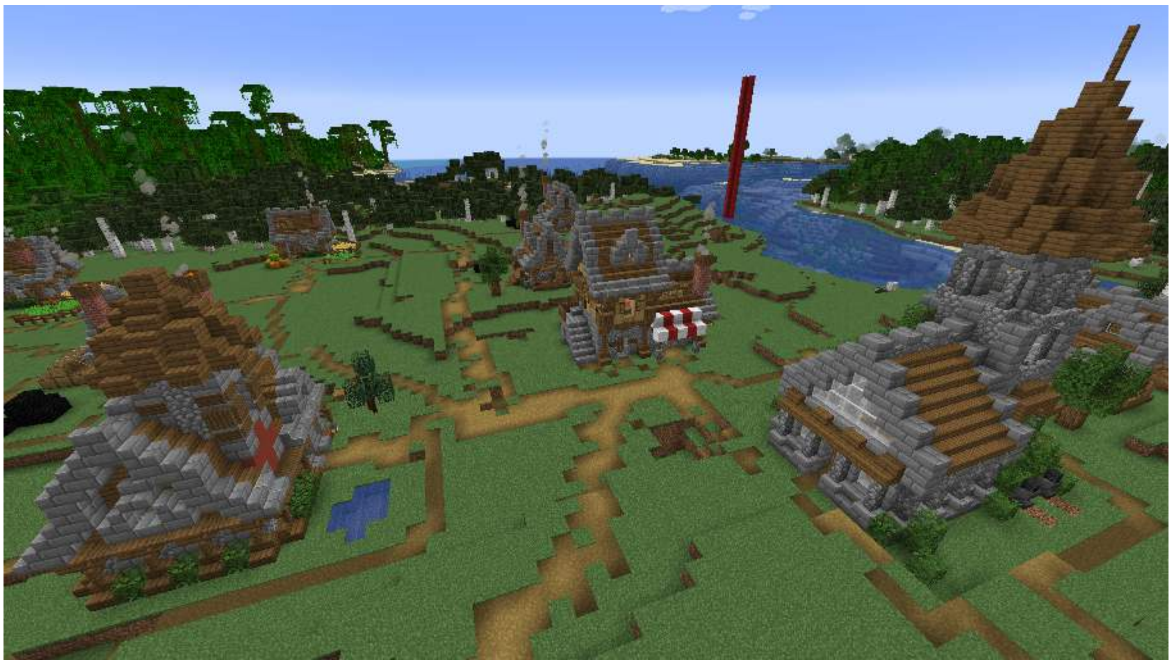
Figure 4.17.: The village generated by the algorithm.

We will now take a closer look at individual parts of the village. Let us start with the village center, which can be seen in Figure 4.18. The three buildings with a function (besides farms) are all next to each other and directly connected by paths. The hospital is the building on the left with the red cross, the church is the one on the right with the high tower, and the tavern is the one in between with the red and white canopy. Inside the hospital is the job site block for a librarian (closest to a doctor), in the church that of a cleric, in the tavern that of a butcher, and next to the farms that of a farmer. If we were to actually place villagers in this village, they would take on the professions just mentioned[36]. The paths between them are wide, and if we compare this with Figure 4.15(d), we can see that this is indeed where most of the pheromones are. Therefore, the village center is not only the geometric center of all houses, but actually the place in the village with the most traffic. There are five normal homes in the immediate vicinity of these three buildings: behind the hospital, to the left behind the tavern, to the right behind the church, and two under the camera out of sight (compare Figure 4.17). Paths lead away from the village center in all directions, connecting all parts of the village.

Next, we examine the paths in the village in more detail. Remember that we had criticized that the paths in default Minecraft villages are all completely straight, doe not respect the terrain, and do not follow any hierarchy. We have solved all these problems and want to show this with examples. Starting with the straight paths, we could already see in Figures 4.14(d) and 4.15(d) that villagers find very organic paths that are very rarely completely straight over a long distance. We can now also see this explicitly in Figures 4.17, 4.19(a), and 4.19(b). No path runs in a perfect line through the village.

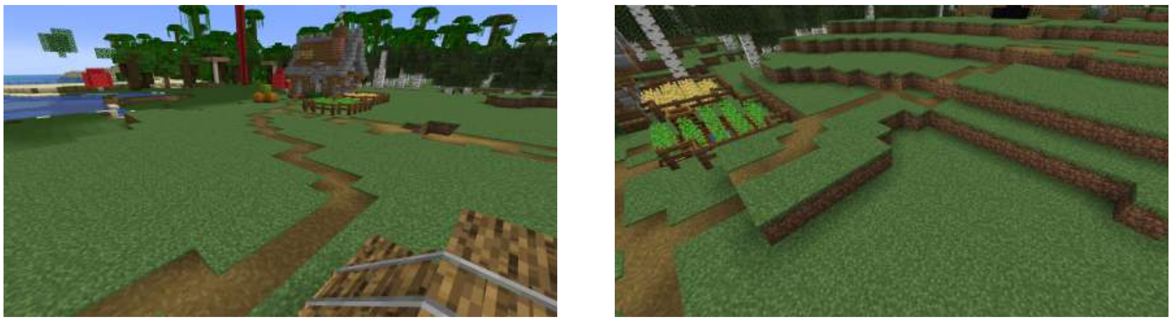
If we compare Figures 4.18 and 4.19(a), we can also clearly make out the different types of paths that together form a hierarchy. The paths in and near the center of the village are very wide. However, a path that goes to a farm at the edge of the village, as in Figure 4.19(a), is very narrow.

[36]Note that we do not place any villagers as this is not trivial, even with Amulet.



Source: Author's image.

Figure 4.18.: The center of the generated village.



Source: Author's image.

(a) A path to a farm.

Source: Author's image.

(b) A path up a hill.

Figure 4.19.: Various paths in the generated village.

Figure 4.19(b) demonstrates well the effect of the villagers' exhaustion. When the villagers walk up the hill, i.e., from the bottom left of the image to the top right, they walk up where the distance between two y-levels is greater. They could just as well go up a little further to the right, but they decide against it for two reasons. They would either have to climb too many times within a short distance, or alternatively make a hairpin turn, which would lengthen their path. Instead, the villagers begin the climb up the hill a few blocks earlier and avoid both.

5. Evaluation

In this chapter, we will evaluate the result of this research. What we cannot do is say with complete certainty that we have clearly generated better (or worse) villages than the default ones in Minecraft. There are two reasons for this. First, we cannot measure the appearance of a village with numbers. While our algorithm is based on ACO algorithms, which is an optimization method, it is not itself an algorithm to find optimal results. Remember that we are not looking for short paths, but natural paths. Second, the definition of naturalness is subjective. While we have explained our ideas of a natural village extensively in Section 2.3.1, other people may not have the same opinion.

Therefore, in order to evaluate our results in a meaningful way, we proceed as follows. First, we show in Section 5.1 that villagers do indeed converge to paths that we have defined as more natural. They find paths that are slightly longer than the direct path, but flatter. After that in Section 5.2, we want to investigate whether, objectively speaking, we have actually generated more natural-looking villages than the default ones in Minecraft. For this, we will survey people from around the world and ask them to compare our villages and Minecraft's villages based on their own definition of naturalness. Last, we show the result of GDMC competition in Section 5.3.

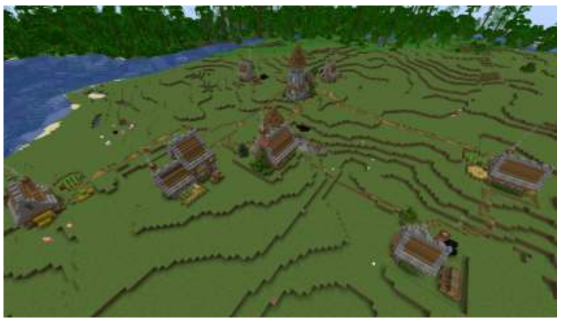
5.1. Path metrics

In this section, we confirm that the algorithm converges to paths that we have defined as natural. In Section 4.4.4, we have already seen that the villagers find short and flat paths, depending on the choice of constants. However, in these experiments, only one villager walked from the same start to the same destination. As we can clearly see from Figure 4.14, the villagers influence each other. Of course, this is exactly what we want, since this forms major paths that are used by many villagers. Therefore, we want to investigate how the properties of the paths change when multiple villagers walk around the same region.

From Figures 4.17 to 4.19(b) of the generated village from Section 4.6, for us the villagers find good paths, even if they all interact with each other. The question naturally arises as to how we can measure how good these paths are. The answer is that we already do. In (4.9), the evaluation of a path is calculated. This value is used to determine how many pheromones a villager's ants secrete along their path. The evaluation in (4.9) depends on exactly the factors by which a natural path is defined, namely length and flatness. We can therefore generate a village and observe how the evaluation (4.9) of the villagers changes over the span of the life simulations.

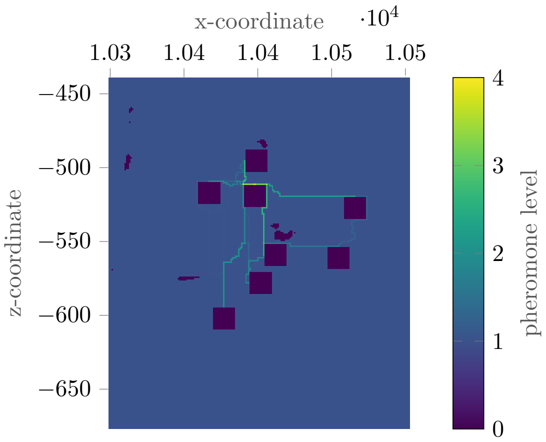
The village that was generated can be seen in Figure 5.1. The parameters are exactly the same as for the village generated throughout Chapter 4. Eight houses with side length $l_H = 15$ are placed per growth iteration, each of the eight villagers is simulated by four ants, $\alpha = 3$, $\beta = 3$, $\gamma = 2$, $H_{\min} = 0.8$, $H_{\max} = 2 - H_{\min} = 1.2$, $s_{\text{recover}} = 4$, $\rho = 0.1$, $r = s_{\text{recover}} = 4$, $\phi = 1$, and $\chi = 2$. The only difference is that the village does not grow.

First, we investigate how the evaluation of the path length changes. For this we only consider the left factor from (4.9). Figure 5.2 shows the average evaluation of the path length over the course of the iterations. The average is calculated over all eight villagers. It makes no difference here that the villagers most likely all walk different paths, which most likely all have different lengths. Recall that for the evaluation of the path length the



Source: Author's iamge.

(a) Overview of the village

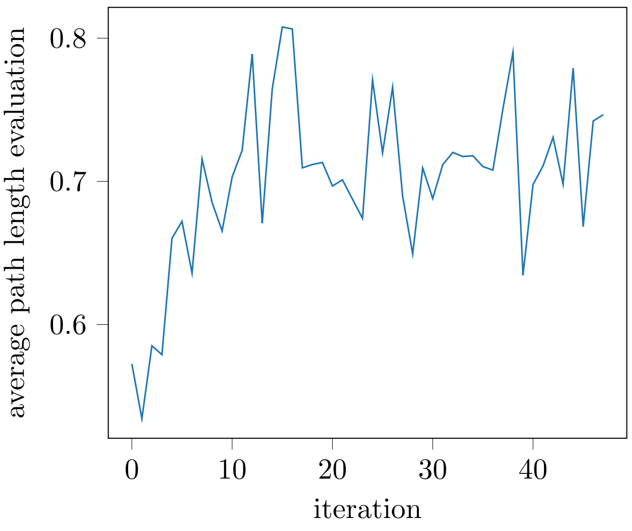


Source: Author's graphic.

(b) The village's paths.

Figure 5.1.: The village which is used to track the evaluation of paths across iterations.

Manhattan distance between start and destination is divided by the actual path length. Hence, all evaluations are normalized to the interval (0,1].



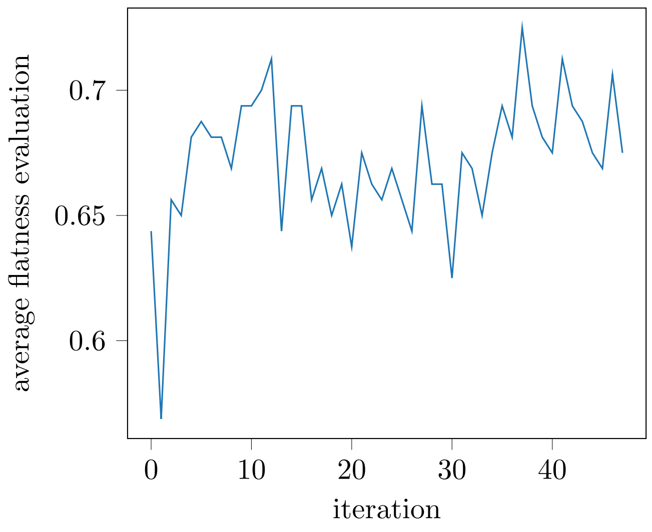
Source: Author's graphic.

Figure 5.2.: The average value of the path length evaluation across life simulation iterations.

We can clearly see that the evaluation increases as iterations pass. In other words, the villagers are finding shorter and shorter paths. This also refutes a doubt that could have been expressed. It could have happened that the different paths negatively influence each other. To be precise, the could have lead the villagers off their path to such an extent that they walk a detour that is far too long. However, this is not the case. Instead, the results suggest that the network of paths formed is constructed in such a way that a villager can walk a short distance from any house to another while staying on the formed paths.

Next, we examine whether the paths also become flatter over time. In other words, the right-hand factor of (4.9) is considered. Recall that this value is closer to one the flatter the path is, and it is closer to zero if the path has at least one bumpy spot. This means that this factor is also normalized to the interval $(0, 1]$ by design, and there is therefore no problem in considering the average value of all villagers. The result is shown in Figure 5.3.

As we can see, the paths also become flatter over time. As with the path length evaluation,

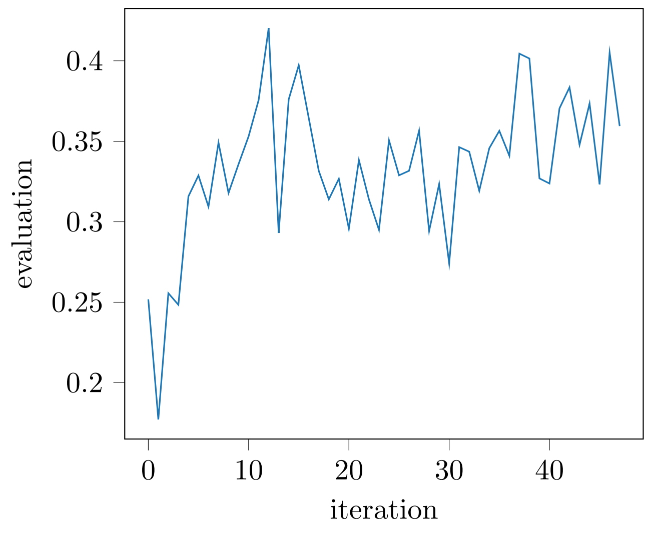


Source: Author's graphic.

Figure 5.3.: The average value of the flatness evaluation across life simulation iterations.

this means that villagers are not unnecessarily walking up and down when they use the path network. This figure also suggests that no path network forms that consists only of short paths or only of flat paths. The villagers find a balance between both criteria, which is exactly the desired result.

The evaluation resulting from (4.9) can be seen in Figure 5.4. Since it is a combination of the two evaluations just shown, it becomes larger over the course of the life simulations, as expected.



Source: Author's graphic.

Figure 5.4.: The average path evaluation across life simulation iterations.

As mentioned earlier, these results confirm that the paths found by the villagers converge to a network that has the desired properties. Although in the above experiment the influence of the path length is generally more weighted than that of the flatness of the path ($\beta = 3$ vs. $\gamma = 2$), the paths become both shorter and flatter over time.

5.2. Human evaluation

As mentioned earlier, the way we perceive the appearance of something is subjective. The criteria by which we, the authors of this thesis, defined the naturalness of villages in Section 2.3.1 are our imaginations of it. Our definition of naturalness may be subconsciously influenced by a number of factors, thus making our judgment about the naturalness of a village in Minecraft biased.

One such factor is the fact that we have a lot of experience with the game. We have seen the villages in Minecraft hundreds if not thousands of times. Furthermore, we have seen many human-built villages and cities whose planning is much more thought out than in the random generation of default villages in Minecraft. Since this gives us a basis for comparison, it is easy for us to claim that the default villages could be more natural. Someone who has little to no experience with the game might not have any complaints about the naturalness of the villages as they cannot imagine how a village might look different.

Another factor that influences us is our origin. The structure of villages and cities often differs greatly depending on where you are on earth. The environment in which a human grows up and spends most of their life is perceived as natural, simply because they are used to it. If somebody grew up in rural Europe, for example, they find placement of houses with plenty of space between them and curvy streets natural. For someone from another part of the world, however, it may seem completely counter-intuitive.

5.2.1. Overview of the survey

For the reasons just mentioned, it is therefore necessary to collect the opinions of many people with different backgrounds in order to make a meaningful judgment about the outcome of this research. For this, we asked 45 people to compare the default Minecraft villages and the ones we generated. We did not tell them which villages were which. Of course, someone who has experience with the game can quickly tell which village is not a default Minecraft village. This is not a problem, however, because it lets us compare how people with playing experience and people without judge the villages. Of the 45 participants, 33 ($\approx$ 73.3%) said they had played Minecraft before. The remaining 12 ($\approx$ 26.7%) have no prior playing experience at all. The participants come from different parts of the world. Of the 33 ($\approx$ 73.3%) participants who are from Europe, almost all are from Germany. The remaining 12 ($\approx$ 26.7%) participants come from different countries in East Asia, most of them from Japan and Taiwan.

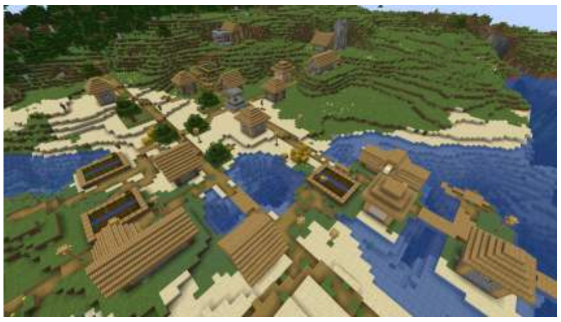
We showed the same eight images to all participants. Four of them are of default villages from Minecraft (see Figure 5.5), and the other four are of villages generated by us (see Figure 5.6). After that, we asked the following three questions (for translations in other languages, see Appendix C):

1. Which group's villages do you find more natural?

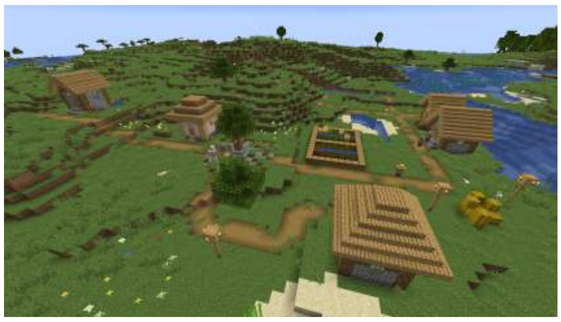
2. Now focus only on the placement of the houses within the villages, i.e., how the houses are distributed, etc. In which group do you find the placement of the houses more natural?

3. Now focus only on the paths within the villages, i.e., how the paths go, etc. (Paths are the light brown lines that go between the houses and through the grass.) Which group do you think has more natural paths?

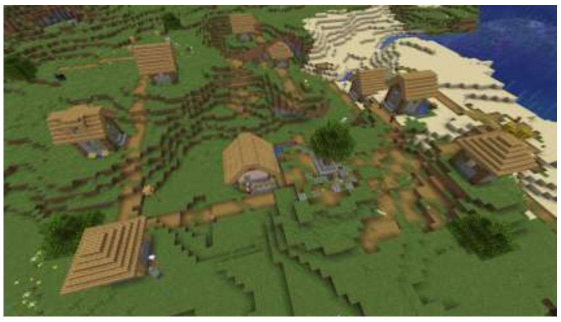
After each answer, participants had to briefly explain why they felt that way. Notice that we do not define the term "natural" in any of the questions or anywhere else in the survey. We did this deliberately because, as we just said, everyone has a different idea of naturalness.


Source: Author's image.

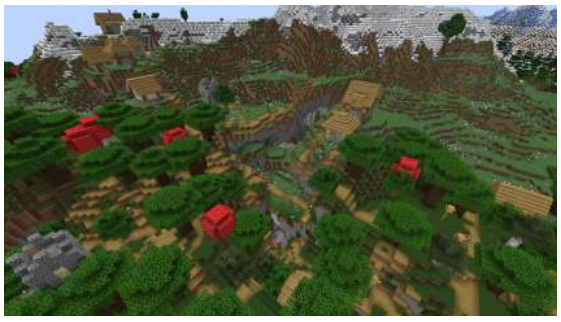
(a) The first village.


Source: Author's image.

(b) The second village.

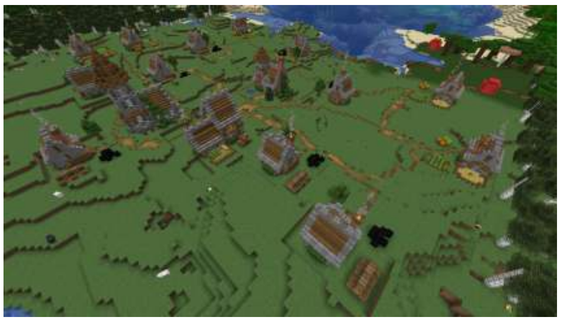

Source: Author's image.

(c) The third village.

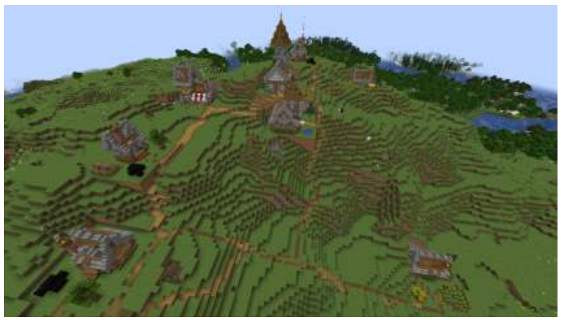

Source: Author's image.

(d) The fourth village.

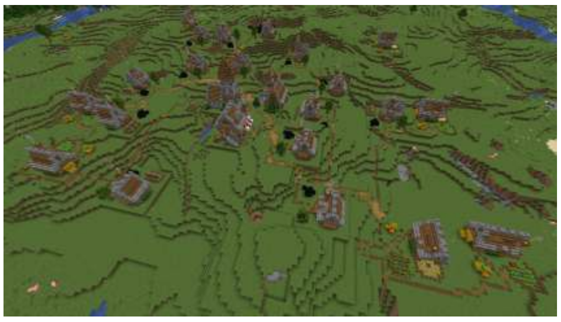
Figure 5.5.: The four default Minecraft villages that were shown to the participants.


Source: Author's image.

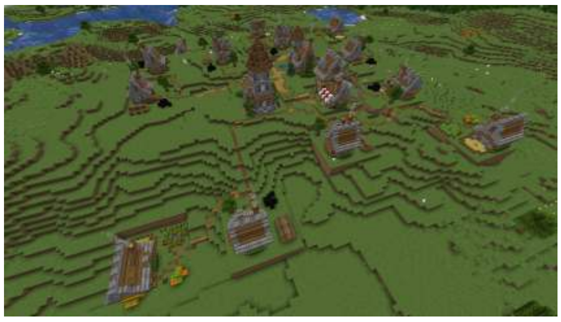
(a) The first village.


Source: Author's image.

(b) The second village.


Source: Author's image.

(c) The third village.

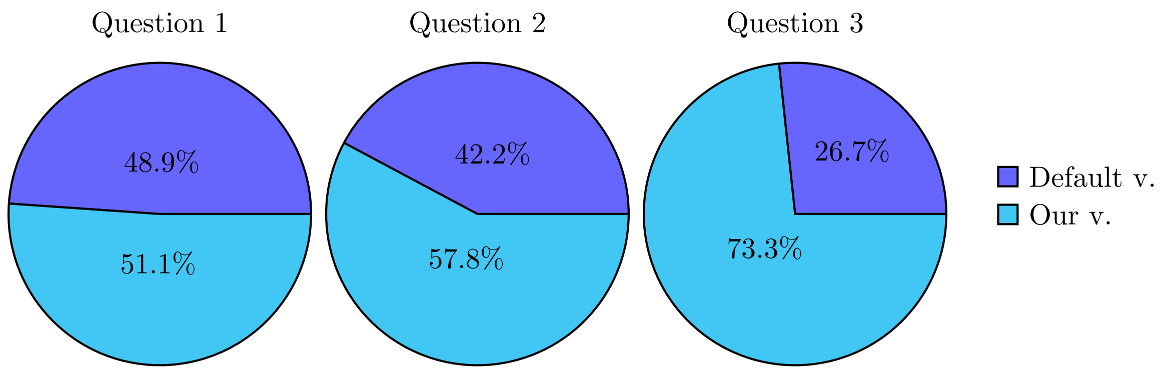

Source: Author's image.

(d) The fourth village.

Figure 5.6.: The four villages generated by us that were shown to the participants.

5.2.2. Overall results

First we will look at the overall results, that is, no distinction is made between participants. The results are shown in Figure 5.7.



Source: Author's graphic.

Figure 5.7.: The overall results of the survey.

As we can see, the opinion about the naturalness of the village as a whole (i.e., question 1) is very divided with 22 votes for the default villages and 23 for our villages. Of the participants who voted for our villages, many say that they find the design of the houses more natural. They are more detailed and seem more realistic and lively. However, some participants who voted for the default villages say that the houses from our village are so detailed that it is difficult to see their function. They perceive these details as disturbing noise, so to speak.

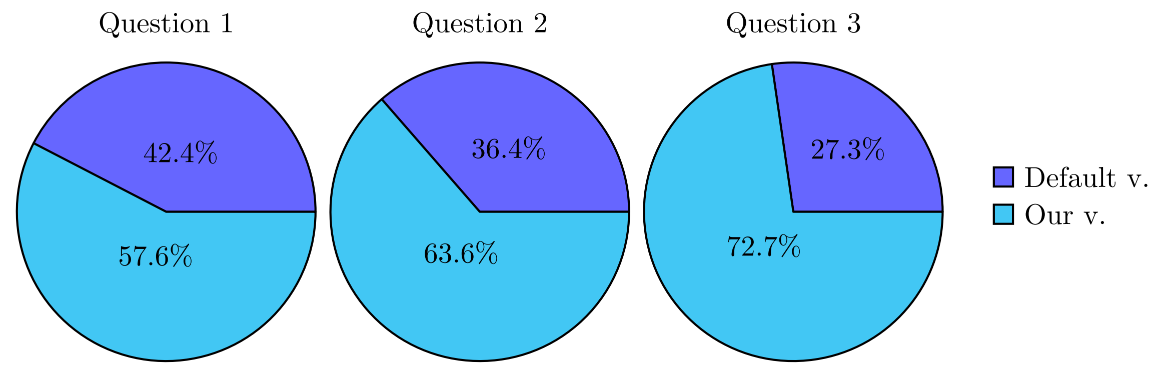
The participants are also somewhat divided on the second question, with 19 to 26 votes. Many of the participants who voted for our villages praise the clearly identifiable, more densely populated village center and the fact that buildings such as the church can be found there, whereas farms are farther outside. The fact that buildings are not placed in the middle of steep slopes or other hard to reach areas are also positively received. The participants voting for the default villages mainly do not like the fact that there is so much space between the houses in our villages.

In contrast to the first two questions, the participants agree on the naturalness of the paths (i.e., question 3). With 33 participants, almost three quarters voted for our villages, and only 12 for the default villages. Participants feel it is more natural that all of our villages' paths are traversable and lead to a destination. The paths are not only straight, but adapt more to the terrain. Also, the paths have different widths and are wider in the center of the village than further outside. Participants who find the paths in the default villages more natural do not like that our paths are sometimes overgrown with grass, think they are too narrow, or criticize the lack of decorative elements such as street lamps.

5.2.3. Results separated by playing experience

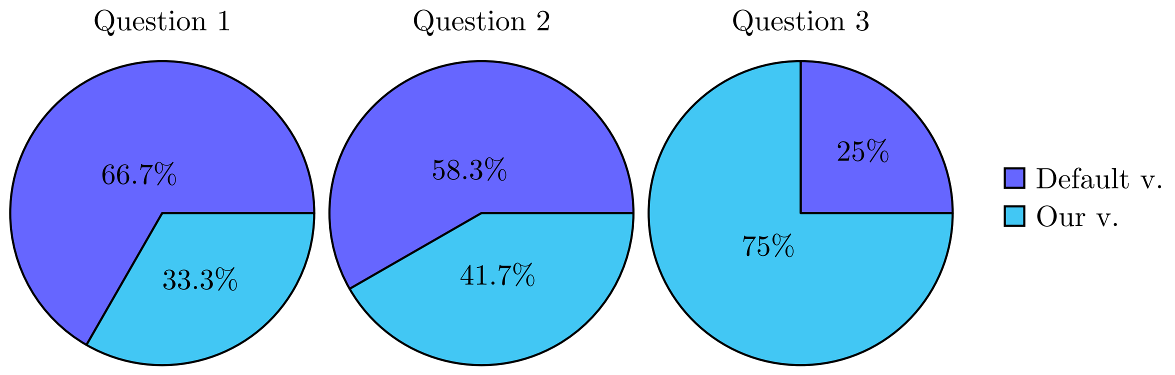
Next, we compare how playing experience affects the perception of naturalness. Figure 5.8 shows the result of the survey separated into participants who have played Minecraft before and participants who have not.

We can see that participants with playing experience voted slightly more in favor of our villages concerning the general naturalness (question 1) and the naturalness of the houses' placements (question 2), and voted almost the same as everyone combined for the naturalness of the paths (question 3). For question 1, we noticed an interesting tendency. Some of the participants with playing experience who voted for the default villages justify doing so with their own definition of a natural Minecraft village. They understand this to mean a village as



Source: Author's graphic.

(a) Participants with playing experience.



Source: Author's graphic.

(b) Participants without playing experience.

Figure 5.8.: The results of the survey separated by playing experience.

it is usually found in Minecraft. This definition makes sense because, as mentioned earlier, many people define as natural what they are used to. It is therefore perfectly understandable that a certain number of experienced players find the default villages more natural.

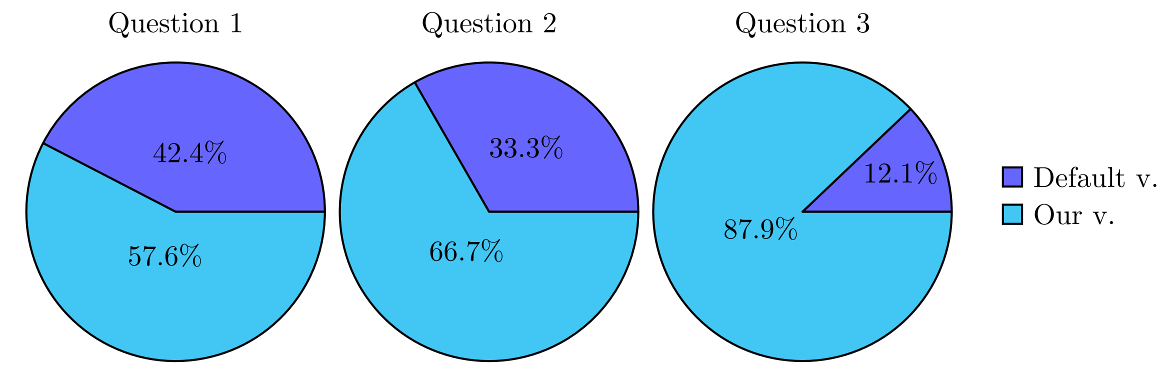
Participants without playing experience, on the other hand, find the default villages and houses more natural. They do not give a concrete reason for their decision. Instead, they say that it just feels more like a village they could imagine living in themselves. The opinion about the paths remains virtually unchanged.

5.2.4. Results separated by participant origin

Finally, the participants are divided depending on their origin. As mentioned earlier, there is one group from Europe and one from East Asia. The results of the survey for each group can be seen in Figure 5.9.

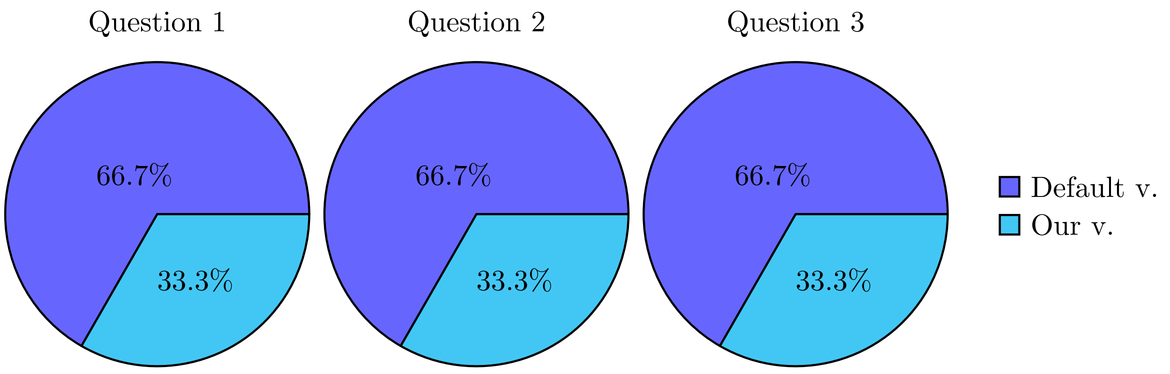
We can see that participants from Europe still prefer our villages. Moreover, two thirds of them find the placement of the houses more natural. Of the participants who find the enough space between buildings to be good, all but one are from Europe. The answers to question 3 show a clear difference to the results before. Of the 33 participants from Europe, 29 find the paths of our villages more natural. They say that they find the route of the paths more logical and that it reminds them more of paths as they know them from real life.

The participants from East Asia, on the other hand, have a completely different opinion. They find the default villages more natural in every respect. We want to elaborate on two of the reasons they give for this. To them, it feels unnatural that the houses have big



Source: Author's graphic.

(a) Participants from Europe.



Source: Author's graphic.

(b) Participants from East Asia.

Figure 5.9.: The results of the survey separated by the origin of the participants.

distances between each other. This impression makes sense when we compare the population density[37] of Europe and East Asia. In 2020, Germany had a population density of 238.5 people per km^2, whereas Japan was at 343.61 people per km^2 and Taiwan at 672.73 people per km^2. This difference becomes even greater when population density is compared only within habitable areas. In 2020, Germany with 83,200,000 inhabitants[38] and a habitable area[39] of 234,582 km^2 had a population density of 354.67 people per km^2, and Japan with 124,947,000 inhabitants[40] and a habitable area of 114,885 km^2 had a population density of 1087.58 persons per km^2. People in East Asia live much closer together than people in Europe. It is therefore perfectly understandable that a participant from East Asia would find it more natural if houses were closer together. They also feel it is unnatural that so much stone is used in the design of the houses. As before, this feeling is based on the familiar environment. For example, modern Japanese houses are mainly made of cement and wood, and traditional houses are pretty much all wood. Accordingly, it is understandable that the extensive use of stones in the design of the houses is perceived as strange.

[37]The data comes from the following website: `https://ourworldindata.org/grapher/population-density?time=2020/`.

[38]The data comes from the following website (page in German): `https://www.destatis.de/DE/Presse/Pressemitteilungen/2021/06/PD21_287_12411.html`.

[39]The data comes from the following website (page in Japanese): `https://raven38.hatenablog.com/entry/2020/08/17/183852/`.

[40]The data comes from the following website (page in Japanese): `https://www.stat.go.jp/data/jinsui/2022np/index.html`.

5.2.5. Discussion

Based on this survey, we briefly discuss whether we have succeeded in generating more natural villages. It is difficult to give a definite answer to this. If we look at the results from Section 5.2.2, we see that the general opinion is almost perfectly balanced. The results from Sections 5.2.3 and 5.2.4 also imply that it depends on a person's background whether they consider a village natural or not. Someone who has played Minecraft before prefers our generated villages, while someone with no playing experience tends to prefer the default villages. Moreover, someone from Europe feels our villages are more natural than someone from East Asia.

Nonetheless, we find that the results speak more than enough in our favor to claim that our villages are more natural. We feel that the most important results are those from Figure 5.8(a). The survey was short and participants who had no prior playing experience with Minecraft could most likely only get a very rough idea of Minecraft and the villages in that short amount of time. An experienced player on the other hand already has the necessary knowledge to make meaningful assessments about things related to Minecraft. We strongly believe that it validates our methods that participants with playing experience find our villages more natural.

Moreover, the focus of this thesis was not the overall appearance of the village, but the placement of houses and paths. That is, the most important result is how the participants perceive these two parts about the villages. Figure 5.8(a) shows that the clear majority of experienced players find these points natural in our villages. This is an evident indication that our method succeeded in finding more natural placements for houses and generating more natural paths.

5.3. GDMC competition result

As mentioned in Section 2.3.2, we have submitted our algorithm to the GDMC competition. At the current time, the results have not yet been made public. As soon as they are, we will add them to this section.

Since the evaluation is done by human judges, this section could also be put under Section 5.2. However, the algorithm used to generate the villages for the survey and the one used to generate villages for the GDMC competition differ enough that it makes more sense to separate them.

6. Conclusion

This chapter provides a concluding overview of the thesis. The main points of the methodology are highlighted and the results are briefly discussed. Finally, ideas and suggestions for future work are proposed.

6.1. Techniques and results

In this thesis, villages in Minecraft were generated with the goal of making them look more natural than the default villages in Minecraft worlds. Naturalness was defined by the position of the houses within the village and by the properties of the path network that connects all the buildings. Houses should be placed on flat ground, and houses with functions should be where it makes sense for them to be. Paths should respect the slope of the terrain and follow a road hierarchy. To achieve this, houses were likely to be placed close to the village center and where the terrain was flat. For the generation of the paths, a method based on ant colony optimization algorithms was developed. Villagers marked their paths while walking through the village, a process they repeated several times. The strength with which a path was marked depended on its evaluation, which in turn was based on how long and how flat the path was. The shorter and simultaneously flatter a path was, the higher its evaluation and the stronger it was marked. Other villagers were more likely to follow a path the more strongly it was marked. Placing houses and generating paths was repeated to make the village grow.

The resulting villages were evaluated in several ways. First, it was analyzed whether the path network found by the villages was actually formed in such a way that it connects one house of the village to every other by a short and flat path. For this, it was considered how the evaluation of the paths changed over the span of the villagers walking through the village. This study showed that the villagers take shorter and flatter paths as time passes, thus confirming our methodology.

Next, the results of a survey were discussed. For the survey, people from different backgrounds were asked to compare Minecraft's default villages and the villages generated by this thesis' algorithm. Moreover, they were asked to name which of the villages they found more natural. The responses to this survey confirmed the assumption that the perception of naturalness is a subjective concept. Nevertheless, the results indicated that the villages generated using ant colony optimization algorithms are indeed more natural than the default villages in Minecraft.

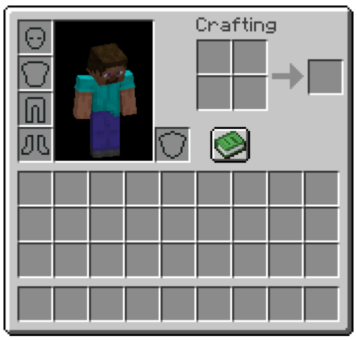
6.2. Future work

Future work includes a more realistic simulation of the villagers. Instead of not assigning a function to them and only giving the houses a function at the very end, this could be done earlier. For this purpose, the profession, residence, and workplace of a villager is determined at the beginning. A villager does not go to a random house, but instead determines their destination depending on their profession and other needs. Most of the time, they move back and forth between their house and workplace, but they also needs to visit regularly, for example, the market and occasionally, for example, the hospital. If the early fixing of a house's function creates a situation where an important building is located far outside, it can exchange places with a "less important" house closer to the center.

Another idea is to use other types of ACO algorithms. The algorithm presented here is based on one of the most basic versions of the ACO algorithms. There are many other methods that build upon it, and many of these are much faster than the basic version. One such is, for example, to combine the basic ACO algorithm with a local search algorithm to find locally optimal solutions. Of course, the notion of an optimal solution would have to be carefully defined, since the goal is not to find the shortest possible paths but the most natural ones.

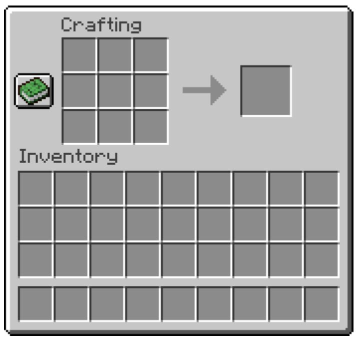
A. Additional information about Minecraft

This appendix collects information about the game Minecraft that we think should be mentioned in this work, but is not essential for the main part.



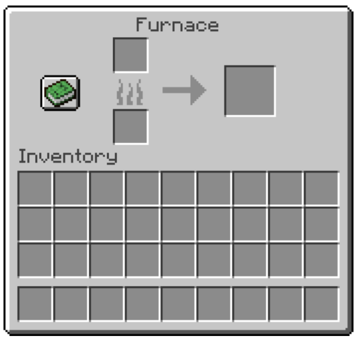
Source: https://static.wikia.nocookie.net/minecraft_gamepedia/images/d/db/Inventory.png/revision/latest?cb=20230128121220.

(a) Inventory interface.



Source: https://static.wikia.nocookie.net/minecraft_gamepedia/images/3/39/Crafting_Table_GUI.png/revision/latest?cb=20220729164204.

(b) Crafting table interface.



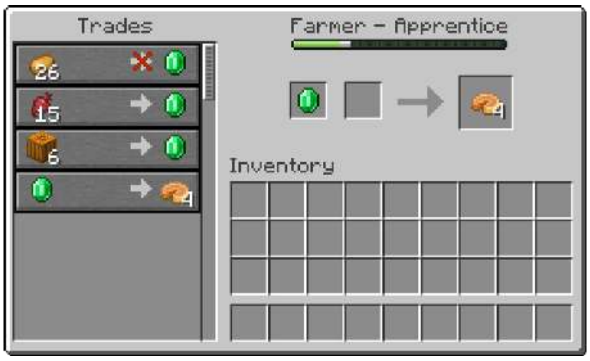
Source: https://static.wikia.nocookie.net/minecraft_gamepedia/images/6/60/Furnace_GUI.png/revision/latest?cb=20190311160314.

(c) Furnace interface.

Figure A.1.: The three main ways to craft in Minecraft. In the inventory, all the materials needed for a craft must be placed in the grid at the top right, and in the crafting table, in the large grid at the top left. Depending on the item that the player wants to craft, certain materials need to be arranged in a certain way. The item will then appear in the field to the right, but will not be crafted until the player clicks on it. The materials on the left are consumed in the process. In the furnace, the fuel goes under the fire icon and the material to be smelted goes above it. Different fuels can smelt different amounts of materials. Unlike crafting, smelting takes some time, and once the process is complete, the new item appears in the field to the right. If the furnace runs out of fuel, smelting is aborted.

The player can make potions that have various positive and negative effects, and enchant their weapons, tools, and armor. Enchanting requires experience, which can be obtained by mining, defeating mobs, or breeding animals, for example [25]. There are two ways to enchant items. The first is at an enchanting table, which additionally requires the mineral lapis lazuli. The second is at an anvil. For this, however, the player needs other already enchanted items or enchanted books, which we mentioned in section 2.3. Since an enchantment can increase the strength and efficiency of an item by a lot, enchanting is an important part of Minecraft.

There are different dimensions in a Minecraft world [25]. The scene from Figure 2.3, the blocks from Figures 2.4(a), 2.4(b), and 2.4(c), and the creeper from Figure 2.5 are all part of the so-called overworld. This is the main dimension in Minecraft and also the one in which the player starts and usually spends most of their time. There are a variety of biomes, such as plains, forest, desert, swamp, beach, river, and ocean, which have different geographical characteristics, flora, temperature, humidity, and environmental colors. The second dimension is the Nether, a dimension full of lava and hostile mobs that the player can reach by building a portal. The third and final dimension is the End, made up of flying islands, where one of the two bosses can be found. It is also reached by a portal [25]. This thesis focuses on the overworld, since we work with objects that can be found exclusively there.



Source: https://static.wikia.nocookie.net/minecraft_gamepedia/images/2/20/VillagerGUI_1.14.png/revision/latest?cb=20190726200924.

Figure A.2.: The trading interface of a farmer villager. On the left, all available trades are listed, and on the top right, the villager's profession and level is shown (apprentice is the second level). The left field to the left of the arrow is either for emeralds or the items the player wants to sell (here one emerald), and to the right of the arrow is the result of the trade (here four pumpkin pies). The items to the left of the arrow are not consumed until the items to the right of the arrow are placed into the inventory. Some trades require additional items besides emeralds, which is why there is a second field to the left of the arrow. Notice the crossed out arrow of the top trade in the list. After repeatedly doing the same trade, its supply is exhausted, and the trade is not available again until the villager has worked on their job site block.

Players can trade with employed villagers in villages. The currency used for this is emeralds, which are very rarely found in the world outside of trading, and the price varies depending on the item. It is only possible to trade with adult villagers who have a profession. An employed villager makes offers to the player depending on their profession and level. When the player trades with a villager, their level increases over time. The higher the level, the more different items the villager can buy and sell [25]. The trading interface can be seen in Figure A.2. Many players trade most often with librarians, as they have enchanted books available, which we have talked about in section 2.2. There are enchantments that cannot be obtained by using an enchantment table. Instead, the player has two options. The first is to explore the world and hope to find an enchanted book with exactly the enchantment they want. However, since enchanted books are very rare to find, this can take a lot of time. The second option, and the one players usually choose, is to place lecterns, the jobsite block for librarians, in villages and find a villager who sells the desired book.

Minecraft has several editions, some of which have been discontinued. *Minecraft: Java Edition* is the original version of Minecraft for Windows, macOS and Linux, developed by Persson. The first full version (1.0.0) was released on November 18, 2011, and the most recent version at the time of writing this thesis is 1.20.1. In addition, there is *Minecraft: Bedrock Edition*, a multi-platform version of the game [25]. We use the Java edition of Minecraft for this research. However, since Amulet, the tool we use to access Minecraft worlds (see Section 2.4), is edition-independent, the methods and results of this research are also independent of the edition used.

B. Algorithm implementation details

For finding trees in the region, we can use differences between the different heightmaps of a Minecraft world. Among these heightmaps is one that ignores leaf blocks and one that ignores liquids. Where the heightmap that ignores leaf blocks and one that includes them differ, there is a tree that we then immediately remove using Amulet. Where the heightmap that includes liquids and one that includes them differ, there is either a water block or lava block. As we remove the trees, we keep track of how the default heightmap changes. It provides information about the highest liquid or block that the player cannot walk through (some blocks, such as flowers, do not block the player's movement). Notice that we also take into account if there is water or lava under a tree while removing it.

The following two equations show how the flatness weight ω_F of each block in the buildable area is calculated. Let $l_H \in \mathbb{N} \setminus 2\mathbb{N}$ and $Y(x, z)$ the y-coordinate of a block (x, z), i.e., its value in the heightmap. The flatness L of a block (x, z) in the buildable area is defined as follows:

$$L\left(x, z\right) \coloneqq \sum_{i=-\frac{l_H-1}{2}}^{\frac{l_H-1}{2}} \left(\sum_{j=-\frac{l_H-1}{2}}^{\frac{l_H-1}{2}} \left|Y\left(x+i, z+j\right) - Y\left(x, z\right)\right| \right) \tag{B.1}$$

In (B.1), we calculate the height differences between each block in the $l_H \times l_H$ neighborhood of (x, z) and (x, z) itself, and then sum them up. The more blocks are at the same height or not too much higher or lower than block (x, z), the smaller this value is. This is a very crude way to measure the flatness of a surface, but it is sufficient for our needs. Since we want to use this flatness to find good places for houses and a smaller flatness is better than a larger one, we have to take the inverse of this value. However, this raises a problem. The bias to place the house in flat areas becomes so large that the influence of the village center, which will be introduced in the next step, is too small. One solution would be to swap the maximum with the minimum flatness instead of taking the inverse and then shift all values in between accordingly. However, this now makes the bias towards flat areas too small. Instead, we add a constant to the denominator of the inverse and multiply the entire fraction by this same constant to reduce the bias a bit. In our experiments, we got good results with 32 as the constant value. Thus, the final formula for the flatness weight ω_F is:

$$\begin{aligned} \omega_F\left(x, z\right) &\coloneqq \frac{32}{L\left(x, z\right) + 32}, && \text{(B.2a)} \\ &= \frac{32}{\sum_{i=-\frac{l_H-1}{2}}^{\frac{l_H-1}{2}} \left(\sum_{j=-\frac{l_H-1}{2}}^{\frac{l_H-1}{2}} \left|Y\left(x+i, z+j\right) - Y\left(x, z\right)\right| \right) + 32}. && \text{(B.2b)} \end{aligned}$$

The second heuristic for calculating the placement weight of a block besides the flatness is the Euclidean distance of the block to the village center, or to be more precise, its inverse. However, we encounter a similar problem as with the flatness weight. Near the center, the bias is too strong. This is not good because blocks very close to the center may be favored even though they have a low flatness weight. The solution is the same as in (B.2b), that is, to introduce a constant. We have found that 16 gives good results. So let (x_c, z_c) be the center of the village. We can now define the center distance weight ω_C of a block (x, z) as:

$$\omega_C\left(x, z\right) \coloneqq \frac{16}{d\left(\left(x, z\right), \left(x_c, z_c\right)\right) + 16}, \tag{B.3a}$$

$$= \frac{16}{\sqrt{\left(x - x_c\right)^2 + \left(z - z_c\right)^2} + 16}. \tag{B.3b}$$

This now allows us to define the placement weight ω_P of a block (x, z):

$$\omega_P\left(x, z\right) \coloneqq \omega_F\left(x, z\right) \cdot \omega_C\left(x, z\right). \tag{B.4}$$

C. The survey questions in other languages

To make the survey more accessible to everyone, we have provided the questions and every other body of text not only in English, but also in German and Japanese. To show that, nevertheless, everyone was asked the exact same questions, we briefly list the three questions in the other two languages. In German, the questions are as follows:

1. Die Dörfer welcher Gruppe findest du natürlicher?

2. Konzentriere dich nun ausschließlich auf die Platzierung der Häuser innerhalb des Dorfs, also wie sie verteilt sind usw. In welcher Gruppe findest du die Platzierung der Häuser natürlicher?

3. Betrachte nun nur die Wege innerhalb des Dorfs, also wie diese verlaufen usw. (Wege sind die hellbraunen Linien, die sich zwischen den Häusern und durch das Gras hindurchziehen.) Welche Gruppe findest du hat natürlichere Wege?

In Japanese, the questions are as follows:

1. どちらのグループの村がより自然だと思いますか?

2. 今回は，村の中での建物の配置（つまり建物がどう分布されるかなど）にだけ注目してください．どちらのグループの建物の配置がより自然だと思いますか?

3. 今回は，村の中の道（つまり道がどのように通っているかなど）にだけ注目してください．（道とは，建物と建物の間や草の間を通る薄茶色の線のことです．）どちらのグループの道がより自然だと思いますか?

Bibliography

[1] Agustin Antonissen and Maria Cristina Riff. “An Ant Based Approach for Generating Procedural Animations”. In: *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*. Vol. 1. 2010, pp. 19–26. DOI: `10.1109/ICTAI.2010.12`.

[2] *Bay 12 Games: Dwarf Fortress*. URL: `https://www.bay12games.com/dwarves/` (visited on July 8, 2023).

[3] Michael Beukman et al. *Hierarchically Composing Level Generators for the Creation of Complex Structures*. arXiv:2302.01561 [cs]. July 2023. URL: `http://arxiv.org/abs/2302.01561/` (visited on July 31, 2023).

[4] Steve Breslin. *The History and Theory of Sandbox Gameplay*. July 2009. URL: `https://www.gamedeveloper.com/design/the-history-and-theory-of-sandbox-gameplay/` (visited on July 9, 2023).

[5] Sebastian S. Christiansen and Marco Scirea. “Space segmentation and multiple autonomous agents: a Minecraft settlement generator”. In: *2022 IEEE Conference on Games (CoG)*. IEEE, Aug. 2022, pp. 135–142. ISBN: 9781665459891. DOI: `10.1109/CoG51982.2022.9893679`. URL: `https://ieeexplore.ieee.org/document/9893679/` (visited on July 31, 2023).

[6] Edsger W. Dijkstra. “A Note on Wwo Problems in Connexion with Graphs”. In: *Numerische Mathematik* 1.1 (Dec. 1959), pp. 269–271. ISSN: 0029-599X, 0945-3245. DOI: `10.1007/BF01386390`. URL: `http://link.springer.com/10.1007/BF01386390/` (visited on July 30, 2023).

[7] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. “Ant system: Optimization by a Colony of Cooperating Agents”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 26.1 (Feb. 1996), pp. 29–41. ISSN: 1083-4419, 1941-0492. DOI: `10.1109/3477.484436`. URL: `https://ieeexplore.ieee.org/document/484436/` (visited on July 31, 2023).

[8] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. Cambridge, MA: MIT Press, 2004. ISBN: 9780262042192.

[9] *Dungeon Keeper Wiki*. URL: `https://dungeonkeeper.fandom.com/wiki/Dungeon_Keeper_Wiki/` (visited on July 8, 2023).

[10] Albin Esko and Johan Fritiofsson. *Multi-Agent Based Settlement Generation In Minecraft*. 2021.

[11] E. Galin et al. “Procedural Generation of Roads”. In: *Computer Graphics Forum* 29.2 (May 2010), pp. 429–438. ISSN: 01677055. DOI: `10.1111/j.1467-8659.2009.01612.x`. URL: `https://onlinelibrary.wiley.com/doi/10.1111/j.1467-8659.2009.01612.x` (visited on July 30, 2023).

[12] Daniel Goldberg and Linus Larsson. *Minecraft: The Unlikely Tale of Markus ”Notch” Persson and the Game That Changed Everything*. First English Language Edition. New York, NY: Seven Stories Press, 2013. ISBN: 9781609805371.

[13] Ben Gothard, Naor Volkovich, and James Clare. *Amulet Editor*. 2021. URL: `https://www.amuletmc.com/` (visited on July 18, 2023).

[14] Michael Cerny Green et al. *Exploring open-ended gameplay features with Micro Roller-Coaster Tycoon.* arXiv:2105.04342 [cs]. May 2021. URL: http://arxiv.org/abs/2105.04342/ (visited on July 8, 2023).

[15] Stefan Greuter et al. "Real-time Procedural Generation of 'Pseudo Infinite' Cities". In: *Proceedings of the 1st International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia.* Melbourne, Australia: ACM, Feb. 2003, pp. 87–94. ISBN: 9781581135787. DOI: 10.1145/604471.604490. URL: https://dl.acm.org/doi/10.1145/604471.604490 (visited on July 30, 2023).

[16] Luis Joshua Gutierrez, Dan Hammill, and Samuel Heaney. *Generated Structures - Minecraft Guide.* June 2022. URL: https://www.ign.com/wikis/minecraft/Generated_Structures/ (visited on July 11, 2023).

[17] Wiwandari Handayani and Iwan Rudiarto. "Dynamics of Urban Growth in Semarang Metropolitan – Central Java: An Examination Based on Built-Up Area and Population Change". In: *Journal of Geography and Geology* 6.4 (Nov. 2014), pp. 80–87. ISSN: 1916-9787, 1916-9779. DOI: 10.5539/jgg.v6n4p80. URL: http://ccsenet.org/journal/index.php/jgg/article/view/39642/ (visited on July 12, 2023).

[18] Peter Hart, Nils Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. ISSN: 0536-1567. DOI: 10.1109/TSSC.1968.300136. URL: http://ieeexplore.ieee.org/document/4082128/ (visited on July 30, 2023).

[19] Adam Hayes, Khadija Khartit, and Katharine Beer. *What Are Tertiary Sectors? Industry Defined, With Examples.* Jan. 2022. URL: https://www.investopedia.com/terms/t/tertiaryindustry.asp (visited on July 12, 2023).

[20] Ari Iramanesh and Max Kreminski. "AgentCraft: An Agent-Based Minecraft Settlement Generator". In: (2021).

[21] George Kelly and Hugh McCabe. "Citygen: An Interactive System for Procedural City Generation". In: (Nov. 2007). URL: https://www.citygen.net/files/citygen_gdtw07.pdf (visited on July 30, 2023).

[22] Søren Kjeldsen-Kragh. *The Role of Agriculture in Economic Development: The Lessons of History.* OCLC: ocm85843057. Copenhagen, Denmark: Copenhagen Business School, 2007. ISBN: 9788763001946.

[23] Charles M. Macal and Michael J. North. "Tutorial on Agent-based Modeling and Simulation". In: *Proceedings of the Winter Simulation Conference, 2005.* Orlando, FL: IEEE, 2005, pp. 2–15. ISBN: 9780780395190. DOI: 10.1109/WSC.2005.1574234. URL: http://ieeexplore.ieee.org/document/1574234/ (visited on July 30, 2023).

[24] Stephen Marshall. *Streets & Patterns.* First. OCLC: ocm54778520. London, United Kingdom; New York, NY: Spon, 2005. ISBN: 9780415317504, 9780203589397, 0415317509, 0203589394.

[25] *Minecraft Wiki.* URL: https://minecraft.fandom.com/wiki/Minecraft_Wiki/ (visited on July 8, 2023).

[26] Steve Nebel, Sascha Schneider, and Günter Daniel Rey. "Mining Learning and Crafting Scientific Experiments: A Literature Review on the Use of Minecraft in Education and Research". In: *Journal of Educational Technology & Society* 19.2 (2016), pp. 355–366. ISSN: 11763647, 14364522. URL: http://www.jstor.org/stable/jeductechsoci.19.2.355 (visited on July 8, 2023).

[27] Yoav I. H. Parish and Pascal Müller. “Procedural Modeling of Cities”. In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. ACM, Aug. 2001, pp. 301–308. ISBN: 9781581133745. DOI: 10.1145/383259.383292. URL: https://dl.acm.org/doi/10.1145/383259.383292 (visited on July 30, 2023).

[28] PCGamesN. *How many Minecraft players are there?* June 2020. URL: https://www.pcgamesn.com/minecraft/minecraft-player-count/ (visited on July 9, 2023).

[29] P. Petrović. *Die Kehre im Gebirgsstraßenbau [The Bend in Mountain Road Construction]*. de. Vienna, Austria: Springer Vienna, 1967. ISBN: 9783211808290, 9783709179567. DOI: 10.1007/978-3-7091-7956-7. URL: http://link.springer.com/10.1007/978-3-7091-7956-7 (visited on July 13, 2023).

[30] Dawid Połap. “Designing mazes for 2D games by artificial ant colony algorithm”. In: *Proceedings of Symposium for Young Scientists in Technology, Engineering and Mathematics*. 2016, pp. 63–70.

[31] Aldo Rossi and Peter Eisenman. *The Architecture of the City*. Oppositions books. Cambridge, MA: MIT Press, 1982. ISBN: 9780262181013.

[32] Christoph Salge et al. “Generative Design in Minecraft (GDMC), Settlement Generation Competition”. In: *Proceedings of the 13th International Conference on the Foundations of Digital Games*. arXiv:1803.09853v2 [cs]. Aug. 2018. DOI: 10.1145/3235765.3235814. URL: http://arxiv.org/abs/1803.09853/ (visited on July 13, 2023).

[33] Rommel Dias Saraiva et al. “Using Ant Colony Optimisation for map generation and improving game balance in the Terra Mystica and Settlers of Catan board games”. In: *International Conference on the Foundations of Digital Games*. Bugibba, Malta: ACM, Sept. 2020. ISBN: 9781450388078. DOI: 10.1145/3402942.3409778. URL: https://dl.acm.org/doi/10.1145/3402942.3409778 (visited on July 31, 2023).

[34] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games*. Computational Synthesis and Creative Systems. Cham, Switzerland: Springer International Publishing, 2016. ISBN: 9783319427140, 9783319427164. DOI: 10.1007/978-3-319-42716-4. URL: http://link.springer.com/10.1007/978-3-319-42716-4 (visited on July 18, 2023).

[35] Asiiah Song and Jim Whitehead. “TownSim: Agent-based city evolution for naturalistic road network generation”. In: *Proceedings of the 14th International Conference on the Foundations of Digital Games*. San Luis Obispo, CA: ACM, Aug. 2019. ISBN: 9781450372176. DOI: 10.1145/3337722.3341852. URL: https://dl.acm.org/doi/10.1145/3337722.3341852 (visited on July 30, 2023).

[36] Julian Togelius et al. “What is Procedural Content Generation?: Mario on the borderline”. In: *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*. Bordeaux France: ACM, June 2011. ISBN: 9781450308724. DOI: 10.1145/2000919.2000922. URL: https://dl.acm.org/doi/10.1145/2000919.2000922 (visited on July 18, 2023).

[37] Xbox Wire. *Minecraft Franchise Fact Sheet*. Apr. 2021. URL: https://news.xbox.com/en-us/wp-content/uploads/sites/2/2021/04/Minecraft-Franchise-Fact-Sheet_April-2021.pdf (visited on July 9, 2023).

[38] Joshua Wolens. “’We win… and it’s your fault!’—Dwarf Fortress hits almost half a million sold in under a month”. In: *PC Gamer* (Jan. 2023). URL: https://www.pcgamer.com/we-win-and-its-your-faultdwarf-fortress-hits-almost-half-a-million-sold-in-under-a-month/ (visited on July 8, 2023).