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

Tobias Deinböck[1,2], Chu-Hsuan Hsueh[1][0000−0001−8888−3116], and Kokolo Ikeda[1]

[1] Japan Advanced Institute of Science and Technology, Nomi, Ishikawa 923-1292, Japan
`{hsuehch,kokolo}@jaist.ac.jp`
[2] Centrum Wiskunde & Informatica, Science Park 123, 1098 XG Amsterdam, The Netherlands
`tobias.deinboeck@cwi.nl`

Abstract. Procedural content generation has been widely deployed to automatically generate digital content with limited or indirect user input. This paper shows how ant colony optimization algorithms, a multi-agent system usually applied to shortest-path optimization problems, can be adapted to generate natural-looking villages in the video game Minecraft. To achieve this, houses are stochastically placed in a specified region of a Minecraft world, favoring flat and central areas. Next, villagers inhabiting these houses are represented by multiple ant agents each and simulate life in the village by wandering between houses. The shorter and flatter a found path between two houses is, the more likely the path will be followed by future ant agents. After some iterations, a natural path network connects all houses. This process of placing houses and connecting them with paths can be repeated, allowing the village to naturally grow. Survey results show that the generated villages are perceived as more natural than the default ones, both regarding house placements and path trajectories.

Keywords: Procedural content generation · Multi-agent systems · Ant colony optimization algorithms · Minecraft · Villages · Natural-looking.

1 Introduction

For many years, games have been used as test beds for artificial intelligence (AI) techniques. Procedural content generation (PCG), one such AI technique, is a method of algorithmically creating digital content (e.g., maps, textures, items, music) with limited or indirect user input [17]. PCG sees a lot of application in video games, where it allows developers to obtain a large number of objects or other types of content in a short time. With video games becoming larger and

[†] This research was conducted while the first author was affiliated with the Japan Advanced Institute of Science and Technology.

more complicated, the need for such methods also increases since it has become unfeasible for humans to create every part of a game themselves. As a result, this area has received more and more attention in academic research over the past few years [11]. Among the content to be generated, two of the important targets are villages and cities, which play major roles in many video games.

For creating villages and cities convincingly, it makes sense to consider how they come into existence in the real world. People settle down at some advantageous place and establish (not necessarily paved) paths where they often walk, thus creating an organic and dynamic network of paths connecting every house in the village. New houses are built, and the process repeats. To generate villages, one could therefore try to replicate this procedure. The challenging part is doing this in such a way that the village looks natural, i.e., it looks like it has been constructed by real humans. A natural way to approach specifically the path generation is to employ a multi-agent system (MAS) [19], which is a collection of agents communicating with each other to achieve better individual and collective goals. Generally speaking, an agent is an entity that is situated and acts autonomously in some environment. Note that an agent does not necessarily know about the overall goal, which in this case is generating a path network.

One specific class of MASs are ant colony optimization (ACO) algorithms [3], where many small ant agents (hereafter just "ants") move between two points (individual goal). Each ant has its own decision-making ability but is influenced by the routes of previous ants. More specifically, a shorter route is better and more likely to be followed by 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 and the shortest possible path between the two points (collective goal). In other words, a structure in the larger system is formed by many small individuals. This is similar to humans establishing paths where they often walk, which motivated us to apply ACO algorithms to generate a path network between houses. However, since we are not interested in the shortest but the most natural-looking paths, we incorporate more factors (e.g., adaption to the terrain) into the evaluation of a path's quality.

As a test bed, we use the video game Minecraft[3] (Mojang Studios, 2011). Minecraft's procedurally generated worlds are made of simple blocks and include villages comprising buildings and paths. These villages are also procedurally generated and play an important role in the game as they provide the player with valuable resources. However, the villages do not look very natural[4], which is problematic since it breaks the game immersion. As a result, annually since 2018, the Generative Design in Minecraft (GDMC) AI Settlement Generation Challenge has been held at conferences like the IEEE Conference on Games, where the algorithm that generates the most natural-looking villages wins [14].

[3] Official website: https://www.minecraft.net/.

[4] The definition of "naturalness" is subjective. The work of this paper reflects the authors' views on what a natural village should look like. For the survey later in this paper, the authors' definition of naturalness was deliberately left out so that participants could decide on their own what they perceive as natural.

Specifically, we find that the placement of the houses within the village seems random, that the paths do not respect the slope of the terrain, and that the paths also do not follow any hierarchy. To fix this, we will place houses in flat and central areas and then use ACO algorithms, where the ants find paths between houses that are not too long and not to bumpy. In the end, central houses will be assigned special functions.

2 Literature Review

The procedural generation of villages and cities outside of Minecraft has been studied before. In [6], houses are procedurally generated, but the road network is a simple grid. The roads in [13] are generated using L-systems [10], but they do not adapt to the terrain. Houses are not assigned a purpose in [6, 13], and neither of the cities can grow after their initial generation. In [5], a road is generated that is as short as possible while still respecting the terrain. This is very similar to our ideas, but instead of a whole network only one road is created. Agent-based modeling (ABM) is employed in [16] to simulate the life of town inhabitants, and roads or houses are placed depending on their activities in certain areas. This is different to our MAS because we do not focus on the details of each individual but rather observe all individuals as a whole. Furthermore, roads are placed without any regard to the terrain and houses have no functions.

Though initially intended for shortest-path optimization, ACO algorithms have seen some applications in PCG (not in Minecraft). In [15], they are used to generate maps for the board games Terra Mystica and Catan. To our knowledge, we are the first using ACO algorithms for village and city generation.

There has also been research on village generation in Minecraft before. The approaches in [1, 9] show similarities to [16] in that ABM is used to simulate the villagers' lives. These were also the first methods we saw that allow the village to naturally grow. However, paths in [1] do not adapt to the terrain, and houses in both approaches have no functions. Also, some buildings in [9] may not be connected to the path network. In [8], the house placement is interpreted as an optimization problem where placing houses in favorable areas is rewarded, and paths between them are generated using A* [7]. This approach relies on reshaping the terrain at first, however, and the village does not naturally grow. As far as we know, we are the first to procedurally generate villages in Minecraft that meet all the aforementioned criteria.

3 Methodology

This section goes into detail about how our algorithm generates villages in Minecraft, a game made of procedurally generated worlds consisting of blocks. A high-level overview is given in Algorithm 1, where we take a Minecraft world and a bounding box as the inputs, and generate a village within that bounding box. Note that we only directly interact with the Minecraft world in lines 1 and 12 of

Algorithm 1 when we get the data, remove trees, and construct the village, i.e., we do not really place houses or paths during the simulation steps.

Algorithm 1: High-level overview of the village generation.

```
   Input: A Minecraft world and a bounding box.
   Output: A Minecraft world with a village within the given bounding box.
 1 Load the world and prepare the area within the bounding box;
 2 while time limit not reached do
 3     if first iteration then
 4         Initialize the village;
 5     end
 6     else
 7         Update the village;
 8     end
 9     Populate the village;
10     Simulate life;
11 end
12 Construct the village;
```

3.1 Loading, Analyzing, and Deforesting the World

The first step is to load and prepare the world (line 1 of Algorithm 1). This includes removing trees from the bounding box and noting where liquids (water and lava) are. Trees severely limit space for building houses, which is why it is easier to get rid of them. Houses also should not be built over liquids, so their positions are recorded to later avoid them.

3.2 Initializing the Village

The final goal of the village initialization (line 4 of Algorithm 1) is to find where houses can be built within the bounding box. First, all connected areas are found. Two blocks belong to the same connected area if, and only if, it is possible to traverse from one block to the other while (i) at every step only moving exactly one block in one of the four cardinal directions (i.e., the von Neumann neighborhood [12]), and (ii) never going up or down more than one block. This is very similar to the rules villagers obey when they move later. Every area found except the largest (by number of blocks in that area) is disregarded for any future step, which ensures that everything takes place in an area that villagers can actually reach. This area is further restricted to ensure that a house with an $l_H \times l_H$ floor plan, $l_H \in \mathbb{N} \setminus 2\mathbb{N}$, has at least one block of space between itself and the edge of the bounding box, other connected areas, and liquids (three blocks for lava). This so-called buildable area guarantees that a house's door is always reachable when moving by rules (i) and (ii) above.

3.3 Populating the Village

Next, the first houses are placed into the village (line 9 in Algorithm 1). In the first iteration, the village center is defined to be the block at the average x- and z-coordinate[5] of the buildable area and does not necessarily need to be within the buildable area itself (e.g., in a lake or another connected area). Before any house placement, the buildable area is further restricted to a square with side length $\lfloor n_H \cdot l_H / 2 \rfloor$ around the village center to prevent houses from being placed too far out, where $n_H \in \mathbb{N}$ is the number of houses to be placed per growth cycle. To place a house, a block in the buildable area is stochastically chosen to be the house's center. Flat areas are assigned a higher flatness weight, and blocks close to the village center are more likely to be selected. These two metrics combine to a block's placement weight, i.e., the likelihood of a house being placed on it. The house's door is determined to be on the side with the most blocks belonging to the buildable area in an $l_H \times l_H$ square directly in front of it. Ties are broken randomly. Similarly to the initialization before, the buildable area is updated such that any possible future house has at least one block of space between itself and the newly placed house. This process is repeated a total of n_H times or until no more houses can be placed.

3.4 Simulating Life

The main part of the proposed algorithm is the life simulation of the villagers in the village (line 10 of Algorithm 1). As aforementioned, we are less concerned with every detail of a villager's life, but rather in how they move around the village for placing paths later. An overview can be seen in Algorithm 2.

Algorithm 2: Simulating life.

```
   Input: The number of simulation cycles $n_S \in \mathbb{N}$.
 1 repeat $n_S$ times
 2     foreach house do
 3         Randomly choose another house;
 4         Let a villager wander from the first house to the second;
 5     end
 6     Fade old paths;
 7     foreach villager that has walked between two houses do
 8         Mark the traversed paths, i.e., secrete pheromones;
 9     end
10 end
```

[5] In Minecraft, y is the vertical coordinate (height).

Choosing a House to Wander to. In line 2 of Algorithm 2, a loop is entered where first, for every house, another different house in the village is randomly chosen. This is done randomly since, at this point, houses have no functions, and 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. Also, it naturally allows many villagers to traverse the village center, as desired.

Wandering Between Houses. Next, a villager wanders between those two houses (from door to door). The detailed overview is given in Algorithm 3. As indicated earlier, ants will find the way for the villagers. When those ants move, they leave pheromones behind whose strength depends on the found path's quality. Other ants are likely to follow a path that has more pheromones on it. In the end, these pheromone trails will form the path network of the village. However, there are some key differences between classical ACO algorithms and our specific needs. Our ants (i) have many different starts and destinations instead of just one each, (ii) can get exhausted to avoid steep paths, and (iii) secrete pheromones on the nodes (i.e., blocks) instead of the edges since we are not interested from where a block was reached but that it was traversed in the first place. Moreover, the way a path's quality is determined needs to be adapted since we are not interested in the shortest path but the most natural one.

Trying to Find a Path. To find a natural path, an ant enters a loop in line 5 of Algorithm 3. The ant's current position is set to the starting block, and this block is set to be the sequence of blocks visited so far, meaning the sequence now contains a single element. In addition, we introduce a variable $s_C \in \mathbb{N}_0$ 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 $s_R \in \mathbb{N}$ an ant must take to recover from exhaustion. In other words, if $s_C < s_R$, the ant is exhausted, which will have implications for the next part.

The ant enters another loop, which is executed as long as its current position is not the destination block. In line 10 of Algorithm 3, ants can go to a block that (a) is in the von Neumann neighborhood of the ant's current position, (b) is not more than one block lower or higher than the ant's current position, (c) is not inside a house, and (d) has not been visited before. The first two rules combined with the house placement rules imply that an ant can never leave the largest connected area. The last point avoids backtracking for more efficiency. For a block $B = (x, z)$, where $x, z \in \mathbb{Z}$, we denote the set of all blocks satisfying these three conditions as $\mathcal{N}_B$ and call it the neighborhood of B.

Let $x, z \in \mathbb{Z}$, let $B = (x, z)$ be the block where the ant is currently located, and let $\alpha, \beta, \gamma \in \mathbb{R}_{\geq 1}$ be positive real numbers greater or equal to one. The probability of selecting a block $B' \in \mathcal{N}_B$ is:

$$P_B(B') := \frac{(\tau_{B'})^{\alpha}\,(\eta_{B,B'})^{\beta}\,(\theta_{B,B'})^{\gamma}}{\sum_{\tilde{B}' \in \mathcal{N}_B} (\tau_{\tilde{B}'})^{\alpha}\,(\eta_{B,\tilde{B}'})^{\beta}\,(\theta_{B,\tilde{B}'})^{\gamma}}. \tag{1}$$

Algorithm 3: A villager's pathfinding for wandering between houses.

```
Input: The starting block $B_S = (x_S, z_S)$, the destination block
       $B_D = (x_D, z_D)$, where $x_S, z_S, x_D, z_D \in \mathbb{Z}$ and within the bounding
       box, the number $s_R \in \mathbb{N}$ of steps needed to recover from exhaustion,
       and the number $n_F^{\max} \in \mathbb{N}$ of allowed failed attempts.
Output: A sequence $(B_i)_{i \in \mathbb{N}}$ of visited blocks for each ant.
foreach ant do
    $n_F \leftarrow 0$;                                  // number of failed attempts
    $l_P \leftarrow 0$;                                  // path length
    $l_P^{\max} = 4 \cdot d_M (B_S, B_D)$;               // maximum allowed path length
    loop
        $B \leftarrow B_S$;                              // current block
        $(B_i)_{i \in \mathbb{N}} \leftarrow (B)$;       // sequence of visited blocks
        $s_C \leftarrow s_R$;                            // steps since last climb
        while $B \neq B_D$ do
            $B' \leftarrow \text{chooseNextBlock}\big(B, B_S, B_D, (B_i)_{i \in \mathbb{N}}, s_C, s_R\big)$;
            if $B' =$ none then
                $n_F \leftarrow n_F + 1$;
                break
            end
            $B_L \leftarrow B$;                          // last block
            $B \leftarrow B'$;
            $l_P \leftarrow l_P + 1$;
            if $l_P > l_P^{\max}$ then
                $n_F \leftarrow n_F + 1$;
                break
            end
            if $y(B) \neq y(B_L)$ then                   // climbed up or down
                $s_C \leftarrow 0$;
            end
            else
                $s_C \leftarrow s_C + 1$;
            end
            $(B_i)_{i \in \mathbb{N}} \leftarrow (B_i)_{i \in \mathbb{N}} \oplus B$;   // append current block
        end
        if $B = B_D$ then
            return $(B_i)_{i \in \mathbb{N}}$ & continue with the next ant
        end
        if $n_F > n_F^{\max}$ then
            return () & continue with the next ant   // empty sequence
        end
    end
end
```

In the numerator, we calculate the value of block B' and then normalize it with the denominator. With the three constant exponents α, β, and γ, we can influence the impact of the three different heuristics (pheromone level, distance weight, and exhaustion weight) for pathfinding.

The first of these is the pheromone level $\tau_{B'}$ 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 rather on the blocks themselves. The more ants traversed a block in previous iterations and the more recently they did so, the more likely another ant is to choose this block. In the first iteration, the pheromone level of every block is set to one.

The second heuristic is the distance weight $\eta_{B,B'}$ of block B'. This weight is higher the closer a block is to the destination. The idea behind this is that a villager never walks around completely randomly but always tries to walk in the rough direction of their goal. To have a bias of the same magnitude everywhere independent of the ant's current position relative to the start and destination, the weights are normalized to a fixed interval $[H_{\min}, H_{\max}]$ around one, where $H_{\min} \in (0,1)$ and $H_{\max} \in (1, \infty)$:

$$\eta_{B,B'} := H_{\min} + \frac{\left(\hat{\eta}_{B,B'} - \min_{\tilde{B}' \in \mathcal{N}_B}\left(\hat{\eta}_{B,\tilde{B}'}\right)\right)\left(H_{\max} - H_{\min}\right)}{\max_{\tilde{B}' \in \mathcal{N}_B}\left(\hat{\eta}_{B,\tilde{B}'}\right) - \min_{\tilde{B}' \in \mathcal{N}_B}\left(\hat{\eta}_{B,\tilde{B}'}\right)}, \tag{2}$$

where

$$\hat{\eta}_{B,B'} := \frac{d_M\left(B, B_D\right) + 1}{d_M\left(B', B_D\right) + 1}, \tag{3}$$

is the distance weight without normalization. We use the normalized (2) instead of (3) because the weight in (3) is close to one and has too little influence if the ant is very far from the destination. If it is near the destination, the weight in (3) gets too large for blocks close to the destination and too small for blocks farther from it, resulting in the ant largely taking the shortest possible path.

The last heuristic is the exhaustion weight $\theta_{B,B'}$. As explained earlier, we want to simulate the exhaustion of the ants or, in other words, of the villagers. Let $s_C \in \mathbb{N}_0$ be the number of steps the ant has taken since the last climb up or down a block as before, and let $s_R \in \mathbb{N}$ be the number of steps after which an ant is no longer exhausted. The exhaustion weight is defined as:

$$\theta_{B,B'} := \begin{cases} \frac{\min\left(s_C, s_R\right)+1}{s_R+1}, & \text{if } y(B) \neq y(B'), \\ 1, & \text{otherwise.} \end{cases} \tag{4}$$

In other words, if an ant has just recently climbed, we reduce the value of all the blocks for which the ant would have to climb again to reach them. Once it has fully recovered or if it does not climb, there is no such penalty.

When the ant has found its next block, it travels to this block while also remembering where it was before. If the ant had to go up or down a block (line 22 of Algorithm 3), its steps since the last climb are reset to zero or, in other words, it is exhausted. Otherwise, we increase the steps since the last climb by one since the ant was able to recover a bit more, and append its current position to the sequence of all blocks visited so far.

Termination Conditions. We enforce some constraints to ensure that pathfinding ends in a reasonable time with reasonable results. We set the maximum acceptable path length $l_P^{\max} \in \mathbb{N}$ to four times the Manhattan distance d_M between the start and destination blocks. The Manhattan distance is always less than or equal to the shortest possible path length between two blocks since ants move to blocks in their von Neumann neighborhood. If the current path length $l_P \in \mathbb{N}_0$ becomes larger than this value (line 18 of Algorithm 3), we stop the current search. We want the ants to deviate from the shortest path, but they should not wander around too much. Moreover, sometimes the neighborhood of a block is empty when an ant has already visited all the neighboring blocks (line 11 of Algorithm 3). In this case, the ant aborts its search.

For each ant of a villager to find a path, we initialize the number of failed pathfinding attempts $n_F \in \mathbb{N}_0$ to zero (line 2 of Algorithm 3). Each time an ant fails to find a path, this value is increased. If the ant has reached its destination (line 30 of Algorithm 3), it saves all the blocks it has visited on its path and stops, allowing the next ant to search for a path. If it has instead exceeded the maximum number of failed pathfinding attempts $n_F^{\max} \in \mathbb{N}$ (line 33 of Algorithm 3), it also aborts the search, but in this case it forgets all the blocks it visited in its current attempt. In other words, the ant has not found a path. This way the algorithm does not get stuck if an ant repeatedly has trouble finding a path. If neither of these situations is the case, it starts its search from the beginning.

Preliminary Study of Pathfinding Parameters. As with many AI techniques inspired by nature, there are quite a few parameters that have to be assigned values. The most important ones are α, β, and γ from (1). The higher α is, the more the ants follow the already existing pheromone trails. A value that is too low will cause the paths to not converge, and 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. The higher β is, the more often ants follow the shortest path. Although $H_{\min}$ and $H_{\max}$ also influence this, we fix them in the experiments. 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 higher γ is, the less likely ants are to climb if they are exhausted. This is influenced by s_R, so we can also fix this value. For experiments, we employ the world shown in Figure 1.

When β is very large compared to γ, the villager ignores the terrain and takes the shortest path to the destination. We can see this well in Figure 2(a). In fact, this path is exactly as long as the Manhattan distance between start and destination. When γ is very large compared to β, 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 2(b). The path is clearly longer, but it bypasses the obstacles completely. When we want to generate villages, it is important to find a balance between these two constants.

The Number of Ants per Villager. We assign between four and eights ants per villager. While assigning only one is more intuitive, this contradicts the concepts

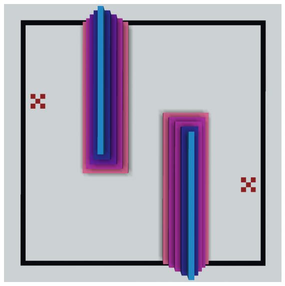


Fig. 1: The world used for the experiments. A villager simulated by four ants walks from the X on the left to the one on the right. The colored blocks are obstacles which become one layer higher with each additional color. The black frame shows the borders (not included) of this 48-by-48 region.

of classical ACO algorithms and does not yield good results in practice. Each ant can be interpreted as a separate day in a villager's life.

Fading Old Paths. Once all villagers have finished Algorithm 3, pheromones evaporate in line 6 of Algorithm 2. This can be interpreted as paths fading because they weather, grass grows over them, and so on:

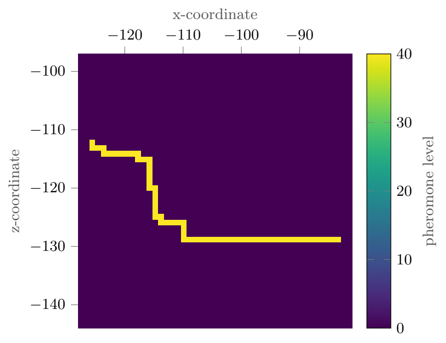
$$\tau_B \leftarrow (1-\rho)\,\tau_B, \tag{5}$$

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

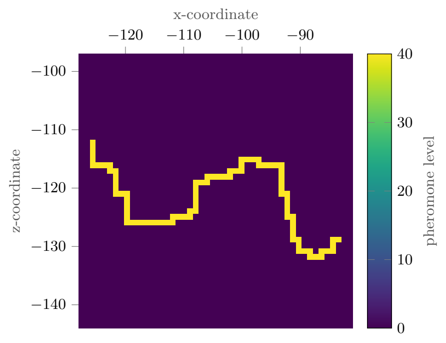
Marking New Paths. As per line 8 of Algorithm 2, it now only remains to secrete pheromones on the newly found paths. Unlike in classical ACO algorithms, we are not interested in the shortest path, but in the one that looks most natural. As aforementioned, a path should not rise and fall too much, but should also not be unnecessarily long. To penalize paths that are too long, we can compare a path to the theoretically shortest path between start and destination (i.e., the Manhattan distance). For penalizing bumpy paths, we can look at its local unevenness. By this we mean how many times an ant has climbed within a contiguous path segment (run). Let $r \in \mathbb{N}$ be the run, that is, the number of blocks over which we want to compute the unevenness, let $\mathcal{B} := (B_i)_{i\in\mathbb{N}}$ be the sequence of blocks the ant has visited along its path, let $k \in \{1,\dots,|\mathcal{B}|-r\}$, and let $\mathcal{B}_r^k := (B_i)_{i\in\{k,\dots,k+r\}} \subseteq \mathcal{B}$ be a sequence of r blocks that the ant has visited in succession. The unevenness S_r on $\mathcal{B}_r^k$ is then defined as

$$S_r(\mathcal{B}_r^k) := \frac{\sum_{k=1}^{r-1} |y\,(B_k) - y\,(B_{k+1})|}{r-1}, \tag{6}$$

where $y\,(B_k)$ is the y-coordinate (height) of B_k for all $k \in \{1,\dots,r\}$ as before. If $\mathcal{B}_r^k$ has fewer than r elements, we calculate the unevenness over all blocks of



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



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

Fig. 2: The influence of β and γ on pathfinding. The values of all other parameters are the same for both cases: $\alpha = 3$, $H_{\min} = 0.8$, $H_{\max} = 2 - H_{\min} = 1.2$, and $s_R = 4$. Furthermore, $\rho = 0.1$, $r = s_R = 4$, $\phi = 0$, and $\chi = 0$.

the path instead and adjust r accordingly. If all blocks are at the same height, the unevenness is zero. However, if two consecutive blocks are always at different heights, then the unevenness is one. In between, it increases linearly.

Now, letting $\phi, \chi \in \mathbb{R}_{\geq 1}$ be positive real numbers greater or equal to one, the evaluation $\varDelta$ of path $\mathcal{B}$ is defined as

$$\varDelta(\mathcal{B}) := \left(\frac{d_M\left(B_S, B_D\right)}{|\mathcal{B}| - 1}\right)^{\phi} \left(1 - \frac{r \cdot \max_{k \in \{1,\ldots,|\mathcal{B}|-r\}}\left(S_r\left(\mathcal{B}_r^k\right)\right)}{1 + r}\right)^{\chi}. \quad (7)$$

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 largest local unevenness is to zero, and tends to $1/\left(1 + r\right)$ otherwise. Now, the ants follow the formula

$$\tau_B \leftarrow \tau_B + \varDelta\left(\mathcal{B}\right), \quad \forall B \in \mathcal{B}, \quad (8)$$

to emit the number of pheromones calculated in (7) on all blocks of their paths.

3.5 Growing the Village

We want the village to grow since 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. First, we update the village center. Since we are no longer in the first iteration, there are houses in the village to calculate it. We then expand the boundaries of the village. In Section 3.3, we had defined the buildable area to be a square around the village center which has a side length of $\lfloor n_H \cdot l_H / 2 \rfloor$, where $n_H \in \mathbb{N}$

was the number of houses to be placed in each growth cycle and $l_H \in \mathbb{N} \setminus 2\mathbb{N}$ was a square house's side length. 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. We use the new village center and $\lfloor n_H \cdot l_H / 2 \rfloor + i \cdot l_H$ as the side length for the buildable area, where $i \in \mathbb{N}$ is the i-th growth iteration we are currently in. 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 move without regard to the buildable area.

Next, we also need to update the placement weights for new houses, which are calculated based on the flatness weights (unchanged) and the village center (most likely changed). In addition, we update the directions of the doors of the already existing houses. As before, we consider the four $l_H \times l_H$ squares directly adjacent to a house for this. This time, however, we want the door on the side with the square with the highest pheromone level since it contains the most developed paths. We do this because a house might be right next to a strong path but turned away from it. Putting the door on the side with the most developed paths prevents villagers from wasting time walking around the house.

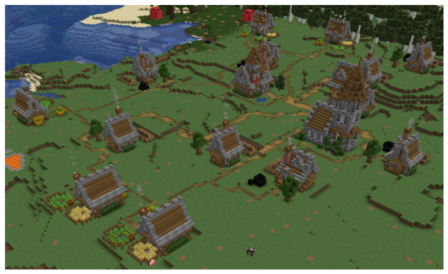
We then normalize the pheromone level by setting the lowest to one, the highest to four, and adjusting all values in between linearly. Otherwise, the pheromone level would be very high on the found paths and extremely small away from them since it decreases exponentially in each simulation cycle following (8), making it extremely difficult for villagers to reach houses that are not yet connected to the network. Since, at the same time, we do not want to get rid of the paths we have found so far, we instead weaken them and at the same time strengthen blocks not on paths.

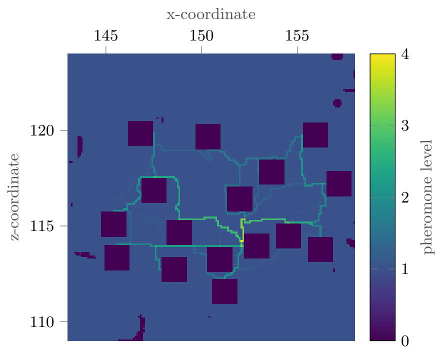
Now, we place houses as in Section 3.3. The only difference is determining the direction of the door, where we look at the pheromone levels, not the number of blocks within the largest connected area. Note that it may happen that houses are placed on paths of the previously found path network.

3.6 Constructing the Village in the Minecraft World

Remember that, until now, nothing has actually been placed inside the Minecraft world yet. Except the tree removal, all simulations have been run on separate data structures. The first step to construct the village, i.e., to place it in the Minecraft world, is to lay all the paths. To be able to distinguish much used paths from little used ones, we normalize all pheromones exactly as before to the interval $[1, 4]$. If the pheromone level is on $[1, 1.2)$, we place no path. If it is on $[1.2, 2)$, the path is little used and we place a path block at that location but do not do anything else. A value on $[3, 4]$ indicates a major path. Instead of placing a path block only at the block that shows this high pheromone concentration, we place path blocks at all blocks in the 3-by-3 square around it as well. This makes the path much wider and shows that it plays a more important role in the village. When the pheromone level is on $[2, 3)$, we do the same, but only place a path block in the 3-by-3 square with a 25% probability for each block within that square (the middle block is still always a path block). This makes it look like the path is in a transition phase between the other two types.

Lastly, houses are placed. To give a house a function, we consider where it is located within the village. In our implementation, there are five types of houses. The building closest to the village center becomes the hospital. The buildings that are second and third closest become the tavern and the church, respectively. The houses that are far out become farms, and all the houses in between become normal homes. An example of a generated village can be seen in Figure 3.


(a) The village.



(b) Pheromone heatmap.

Fig. 3: A village generated by the algorithm. $\alpha = \beta = 3$, $\gamma = 2$, $H_{\text{min}} = 0.8$, $H_{\text{max}} = 2 - H_{\text{min}} = 1.2$, $s_R = 4$, $\rho = 0.1$, $r = s_R = 4$, $\phi = 1$, and $\chi = 2$.

4 Evaluation

4.1 Survey

We asked 45 people[6] to compare the default Minecraft villages and the ones our algorithm generated, and point out which look more natural. 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 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 participants come from different countries in East Asia, most of them from Japan and Taiwan.

[6] Despite Minecraft's large player base, inviting 45 people to evaluate is comparable to or more thorough than previous work (66 participants in [1] and six in [8]).

The same eight images were shown to all participants.[7] Four of them were of default villages from Minecraft, and the other four were of villages generated by our algorithm. While no village generated by our algorithm was placed on exactly the same terrain as a default village, villages from both groups were evenly spread among both flat and hilly terrain for fair comparison. We asked the following three questions: (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 deliberately did not define the term "natural" in any of the questions or anywhere else in the survey because everyone has a different idea of naturalness. The results can be seen in Table 1.

Table 1: Percentage of survey participants that voted for our villages. The values in parentheses are the number of participants in a category.

	Q1	Q2	Q3
Participants with playing experience (33)	57.6%	63.6%	72.7%
Participants without playing experience (12)	33.3%	41.7%	75.0%
Participants from Europe (33)	57.6%	66.7%	87.9%
Participants from East Asia (12)	33.3%	33.3%	33.3%
All participants (45)	51.1%	57.8%	73.3%

The opinion about the naturalness of the village as a whole was very divided with 22 votes for the default villages and 23 for our villages. For the second question, many of the participants who voted for our villages praised the clearly identifiable, more densely populated village center and the fact that buildings such as the church could be found there whereas farms were farther outside. The fact that buildings were not placed in the middle of steep slopes or other hard to reach areas was also positively received. The participants agreed on the naturalness of the paths, i.e., question (3). For a 95% binomial confidence interval [18], the lower bound is about $60.4\% > 50.0\%$, implying statistical significance of the result. The participants felt it was more natural that all of our villages' paths were traversable and lead to a destination. The paths were not completely straight, but adapted more to the terrain. Also, the paths had different widths and were wider in the center of the village than further outside.

We can see that participants with playing experience voted slightly more in favor of our villages concerning the general naturalness and the naturalness of the houses' placements. For question (1), some of the participants with playing experience who voted for the default villages justified doing so with their own

[7] The images can be found in Figures 5.5 and 5.6 in [2].

definition of a natural Minecraft village. They understood this to mean a village as it is usually found in Minecraft. This definition makes sense because, as mentioned earlier, many people define as natural what they are used to.

Finally, we can see that participants from Europe still preferred our villages. The answers to question (3) show a clear difference to the results before. Participants said that they found the route of the paths more logical and that it reminded them more of paths as they know them from real life. The participants from East Asia, on the other hand, found the default villages more natural in every respect. To them, it felt unnatural that the houses had big distances between each other. This impression makes sense when we compare the population density of Europe and East Asia. In 2020, Japan and Taiwan had population densities roughly 1.4 and 2.8 times bigger than Germany, respectively. People in East Asia live much closer together than people in Europe. It is therefore understandable that a participant from East Asia would find it more natural if houses were closer together. Fisher's exact test [4] with $p < 0.05$ shows a statistical significance between the two groups' view on question (3).

4.2 GDMC AI Settlement Generation Challenge 2023

An early version of this algorithm was submitted to be judged at the GDMC AI Settlement Generation Challenge 2023. Overall, it ranked seventh out of eleven participants as the early version was lacking in many design elements that heavily influence the score. However, our villages scored fourth in the adaptability category where judges praised how "the paths mold excellently to the terrain".

5 Conclusion

In this paper, villages in Minecraft were generated with the goal of making them look more natural than the default villages in Minecraft worlds. To achieve this, houses were likely to be placed close to the village center and where the terrain was flat. Villagers marked their paths while walking through the village, a process they repeated several times. The shorter and simultaneously flatter a path was, the stronger it was marked, making other villagers more likely to follow it. This mechanism is similar to ant colony optimization (ACO) algorithms. After some iterations, this formed a network of paths connecting all houses. Placing houses and generating paths was repeated to make the village grow. In a survey, participants were asked to compare Minecraft's default villages and the villages generated by this paper's method. The results indicated that the villages generated using the proposed algorithms are indeed perceived as more natural than the default villages in Minecraft.

Future work includes a combination with agent-based modeling for a more realistic simulation of the villagers. A villager does not go to a random house, but instead determines their destination depending on their profession and other needs. This also means that houses would be assigned a function earlier. Another idea is to use other types of ACO algorithms. There are many other methods that build upon it, and many of these are much faster than the basic version. Furthermore, it is worth doing surveys with larger scales in terms of more participants from different countries, more versatile terrains, and comparisons with other approaches listed in Section 2.

References

1. Christiansen, S.S., Scirea, M.: Space segmentation and multiple autonomous agents: a Minecraft settlement generator. In: IEEE CoG. pp. 135–142 (2022)
2. Deinböck, T.: Procedurally Generating Natural-Looking Villages in Minecraft with Ant Colony Optimization Algorithms. Master's thesis, Japan Advanced Institute of Science and Technology (sep 2023)
3. Dorigo, M., Stützle, T.: Ant Colony Optimization. The MIT Press (jun 2004)
4. Fisher, R.A.: On the Interpretation of χ^2 from Contingency Tables, and the Calculation of P. J. R. Stat. Soc. **85**(1), 87–94 (jan 1922)
5. Galin, E., et al.: Procedural Generation of Roads. CGF **29**(2), 429–438 (2010)
6. Greuter, S., et al.: Real-time Procedural Generation of 'Pseudo Infinite' Cities. In: Proc. 1st Int. Conf. Comput. Graph. Interact. Tech. Australas. South East Asia. p. 87–ff. GRAPHITE '03 (2003)
7. Hart, P.E., Nilsson, N.J., Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Trans. Syst. Sci. Cybern. **4**(2), 100–107 (jul 1968)
8. Huang, S., et al.: Generating Redstone Style Cities in Minecraft. In: IEEE CoG. pp. 1–4 (aug 2023)
9. Iramanesh, A., Kreminski, M.: AgentCraft: An Agent-Based Minecraft Settlement Generator. UC Santa Cruz (oct 2021), https://escholarship.org/uc/item/83p272pq
10. Lindenmayer, A.: Mathematical Models for Cellular Interactions in Development II. Simple and Branching Filaments with Two-sided Inputs. J. Theor. Biol. **18**(3), 300–315 (mar 1968)
11. Liu, J., et al.: Deep learning for procedural content generation. Neural Comput. Appl. **33**(1), 19–37 (jan 2021)
12. von Neumann, J., Burks, A.W., et al.: Theory of Self-Reproducing Automata. University of Illinois Press Urbana (1966)
13. Parish, Y.I.H., Müller, P.: Procedural Modeling of Cities. In: Proc. 28th Annu. Conf. Comput. Graph. Interact. Tech. p. 301–308. SIGGRAPH '01 (2001)
14. Salge, C., et al.: Generative Design in Minecraft (GDMC): Settlement Generation Competition. In: Proc. 13th Int. Conf. Found. Digit. Games. FDG '18 (2018)
15. Saraiva, R.D., et al.: Using Ant Colony Optimisation for map generation and improving game balance in the Terra Mystica and Settlers of Catan board games. In: Proc. 15th Int. Conf. Found. Digit. Games. FDG '20 (2020)
16. Song, A., Whitehead, J.: TownSim: Agent-based city evolution for naturalistic road network generation. In: Proc. 14th Int. Conf. Found. Digit. Games. FDG '19 (2019)
17. Togelius, J., et al.: What is Procedural Content Generation? Mario on the borderline. In: Proc. 2nd Int. Workshop Proced. Content Gener. Games. PCGames '11 (2011)
18. Wallis, S.: Binomial Confidence Intervals and Contingency Tests: Mathematical Fundamentals and the Evaluation of Alternative Methods. J. Quant. Linguist. **20**(3), 178–208 (aug 2013)
19. Weiss, G. (ed.): Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence. The MIT Press, 3. print edn. (2001)