Generative Design in Minecraft Challenge

CO600 Group Project Report

Group M08

Wai Ip Chu, Selina Soo, Ho Kiu Lam, Rhys Davey

wic2, ss2249, hkl9, rad33

School of Computing

University of Kent

Word Count: 6,210

April 8, 2021

Abstract

The objective of the Generative Design in Minecraft Challenge (GDMC) is to create a convincing AI-generated settlement in the video game Minecraft. The result is evaluated via the metrics of adaptability to the terrain, functionality as a settlement in the game itself, reflection of narrative ideas, and pure aesthetics. This technical report explains the approaches taken throughout each stage of the development, describes the difficulties encountered during the process, and finally reflects on what has been achieved, the future direction of the project, and some potential improvements that could be made.

Contents

1 Introduction

1.1 GDMC Competition

The GDMC competition tasks participants with generating a convincing settlement in Minecraft (2011), a procedurally-generated open-world sandbox game. Minecraft's primary feature is the ability of the player to interact with the game's world by breaking and placing the blocks that it is composed of, thus allowing them to construct, modify or destroy various structures.

Auto-generated villages already exist in Minecraft, but they have a number of limitations and bugs. For example, a house may generate at a high elevation, making it difficult to reach. The game attempts to generate a path connecting each building in the village, but these paths do not account for the height of the terrain, which can result in them being interrupted, resulting in an inaccessible building. Moreover, the number of building types is rather limited, meaning there is little variation across different settlements (Salge et al., 2018).

Our solution takes the form of a filter script in MCEdit Unified, a third-party, community-driven Minecraft map editor. This allows us to make use of libraries built for the software, which can modify the terrain of the Minecraft world, to construct our algorithm. In the competition itself, the filter script will be run on three different maps, and the resulting settlement would be evaluated by the four criteria of adaptability, functionality, narrative, and aesthetics. The GDMC judges will rate the performance with a score from 0-10 in each criteria (GDMC, 2021). A score of 0 indicates no consideration has been given to the criteria, a score of 5 is "comparable to a naive human", and a score of 10 demonstrates "superhuman performance" beyond even a group of dedicated experts (Salge et al., 2020).

The key concept of this project is procedural generation, wherein the settlement takes shape as the algorithm adds further assets to it progressively and algorithmically. Due to computational randomness, the generated content varies, even under the same selection and settings. This maintains the uniqueness of each settlement but allows consistent rules to be applied for each new generation. For the purposes of this project, we make use of a combination of hand-made structures stored in schematic files which describe the positions of blocks, and algorithm-generated structures like paths and adjustments to the terrain. This allows us to control the

viability and quality of the settlement while still ensuring it is generated with a degree of randomness.

1.2 Past Submissions

In order to gain a better understanding of the competition, we researched entries from previous years (Salge et al., 2020) (GDMC, 2018). From this we were able to look at the kinds of terrain they were using to evaluate their algorithms, such as islands and mountain ranges, as well as discover the standard that we were competing against. We specifically investigated the highest and lowest scoring entries according to each individual criteria. This allowed us to look at interesting concepts in their algorithms that we may also want to implement or build upon, as well as understand why certain submissions under delivered or failed (corpus/Researches/Past Submission).

We found that the well-performed entries tended to adapt a variety of algorithms in to solve various issues programmatically, such as minimum spanning tree to calculate the smallest number of paths needed and A* algorithm to discover the shortest path between two points. This ensured consistency between each settlement and maintained a standard procedure for applying the generation. As expected, entries with lower scores consisted limited to none consideration in the area of evaluation. For example, they failed to adapt to the terrain, circumventing problems entirely rather than devising solutions. From this, we learned that all of the evaluation criteria are important to the settlement, and that the use of algorithms enhances the quality of the settlement and reduces the workload.

1.3 Objectives

Our primary objectives were to maximise adaptability, functionality, narrative and aesthetics. These four aspects were specified in the competition's evaluation criteria and acted as a cornerstone and guideline for us during development, informing the decisions we made. In terms of adaptability, our algorithm needed to adjust to varied landscapes like uneven terrain, islands, mountain ranges, etc., ensuring that structures are placed on even ground. For functionality, we looked into creating a settlement that would make sense in the real world while considering Minecraft problems such as defence from mobs and food supply. We also wanted our set-

tlement to evoke an intriguing narrative that would make users want to visit and learn more about it simply by looking at the area. Finally, our settlement needed to have a consistent theme and be aesthetically pleasing to look at. Combined, these four objectives should result in a well-executed and successful settlement.

The competition also outlined hardware and runtime requirements and limitations, so we also needed to ensure our algorithm would be capable of executing successfully in line with these requirements while still fulfilling the objectives.

2 Development

2.1 Project Management

Agile project management methodology was utilized for the project, meaning that the entire development progress was divided into multiple life cycles, each containing stages like analysis, development, integration and deployment. Agile is responsive to changes and has a strong focus on interaction between team members, enabling us to alter or add requirements and tasks as needed throughout the development process. For the project, we opted for weekly sprints, as most of the tasks can be fragmented into one week worth of story points and finished within a week. Incorporating techniques like periodic stand-up meetings, short-term sprints and peer development, we were able to interact frequently, ensuring compatibility between different aspects of the algorithm, resulting in a faster development process. Throughout the duration of the project, we experienced a major change to the requirements, when the host introduced additional evaluation criteria for the 2021 entries. Due to the short sprint length and strong communication provided by Agile, we were able to quickly review and evaluate these changes and make decisions on how to accommodate them.

To monitor and log our progress, we also adapted Agile artifacts like the product backlog and sprint backlog, using project management software Jira to record our contributions and manage tasks (corpus/Artifacts/600_Jira.csv). This allowed us to track the progress from different members and for different areas of the project, helping us estimate the development time and resources needed for future sprints. Moreover, it gave us the opportunity to reflect on the progression of each sprint and adjust our plans for the next sprint based on past performance and any

roadblocks that were encountered.

To help us visualise our thought process we found it helpful to create diagrams. This enabled each of us to give feedback to other team members, but also helps the individual more easily talk through their own thoughts (corpus/Proposal). While someone was explaining their logic via an illustration or diagram other members were able to ask questions and suggest alternative solutions to make their proposed approach more efficient. As a result, even though we were individually allocated specific tasks, others were still able to have input.

At the halfway point of our project we critically evaluated the progress we had made, reflecting on the positives and negatives (corpus/Artifacts/600_CtcEva.pdf). From this we identified that the biggest obstacle to our progress was having too many options. After discussing this as a group we concluded that instead of being overwhelmed and slowing down progress by constantly researching, we would narrow down our options and task certain individuals with investigating specific topics further within a given time frame. For example, while looking at how to generate structures we considered schematics, JSON, chunk slices, and several other methods. We eventually narrowed this down to just schematics and JSON, but on reflection we would have saved time by reducing the amount spent researching these.

In order to check the quality of our algorithm, we created an online form to collect user feedback during our project showcase (corpus/Artifacts/600_Feedback.pdf). We invited users to view our settlement and give it a rating from 1 to 5 in the four criteria given by the competition, and comment on their reasoning behind their score. This form allowed us to establish which areas our settlement succeeded in and which it was still lacking in (3.2.1). As a result, we re-prioritized the tasks in our product backlog based on what aspects they were related to, and how frequently those aspects were mentioned in the feedback. For example, at the time of the poster fair certain features were incomplete and not shown in the demonstration. This enabled us to identify what key features visitors spotted that we could add before submission.

2.2 Methodology

To successfully carry out such a large-scale project, we divided the task into several epic stories to tackle. This allowed us to have a better understanding of the targets in each sector, making them easier to manage throughout the development. Each epic story we identified managed a distinct aspect of the development, yet all of them were connected to other stories.

To generate a convincing settlement, space must be created for the structures by removing foliage (2.3.1.1) and flattening parts of the terrain (2.3.1.2). Then, the available space must be divided up (2.3.3.1) and have structures allocated to it (2.3.3.2). Finally, the rest of the infrastructure, such as paths (2.3.3.4) and farms, must be allocated and placed. We categorised and divided this task into interconnected epic stories. The "Terrains" story manages the input data of the area selected, which the landscape in the Minecraft world converts into program readable data. "Edit terrains" manages the processing and manipulation of the data from "Terrains", smoothing the terrain to create further space for the settlement. "City planning" identifies the buildable areas, allocating which places the structures will be generated in. And finally, "Generation" handles the actual placing of those structures.

In terms of the look of the settlement, we opted for a historical concept, taking inspiration from eras in which buildings were mainly made from wood and still had blacksmiths and wells, but with a more 'green' atmosphere that would encourage inhabitants to explore outside. For instance, we wanted to include greenery and foliage in our designs, so we randomly generated trees, small parks, and greenhouses to a more eco-friendly environment. We also made benches available throughout the settlement to entice villagers to go out and relax outside.

To interact with Minecraft's world data, we adapted pymclevel, an external library that MCEdit is based on, which allowed us to manipulate the blocks in the Minecraft world and edit the data programmatically.

2.3 Algorithms and Problems

Studies of the past entries of the competition shows the use of algorithms and a systematic approach enhances the aesthetics, reliability and performance of the settlement, resulting in a higher score in the competition. During development,

we researched an extensive range of different algorithms, covering different topics and fields of study for inspiration or adaptation, from Sobel Operator to flood fill (2.3.1.2), and from Prim's to A* (2.3.3.4). Algorithms related to image processing and statistics were most looked into, as most of the demand was from data processing and graph traversal, both because those are the fundamentals of the project, and also the sections that can be best optimised with an algorithm.

We also encountered several well-known and insufficiently-documented computational problems during development. Polygon covering and packing was investigated the most, as multiple parts of the generation required calculating where to place structures in a consistent and non-overlapping manner. Calculating an efficient layout improves the aesthetic of the generation and reduces conflicts between structures.

2.3.1 Terrains

2.3.1.1 Deforestation / Hazardous Terrain

The first step to process the selection was to clear all the obstacles in the existing selection. These included all the trees and other foliage (bushes, mushrooms, flowers, etc.), as well as hazardous blocks like lava and cactus; these would be a danger to visitors, and safety was specified in the GDMC criteria. In the case of lava, we opted to remove only surface lava pools rather than all of the lava in the map, as inaccessible underground lava would not pose a danger to visitors. To accomplish this, we created a "lava height map" to check for lava blocks on the surface and change them into grass blocks, as shown in figure 1. We were able to remove these blocks quickly and efficiently using chunk slices.

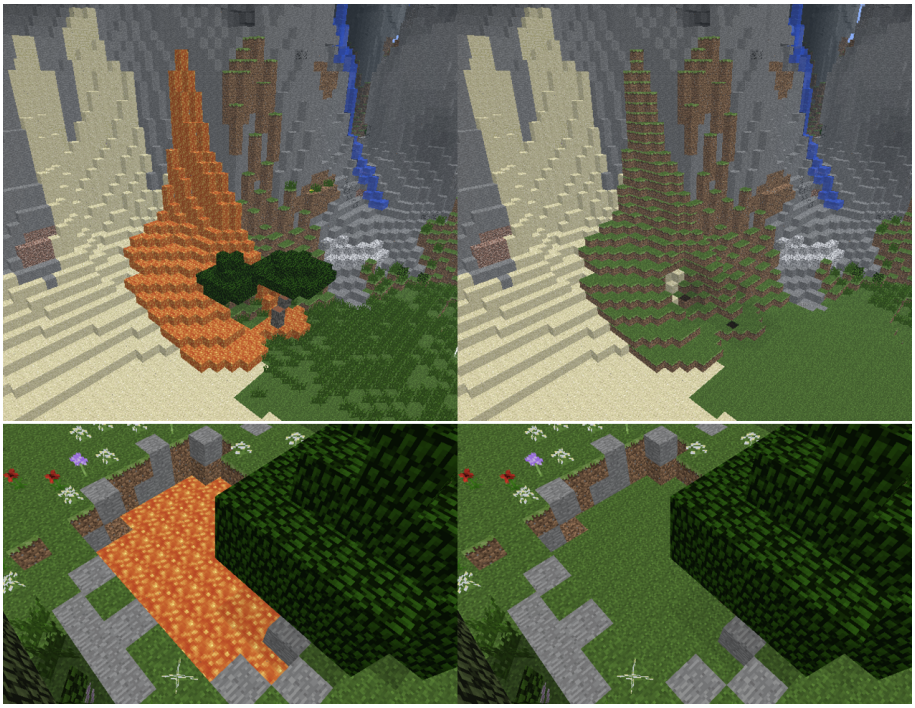

Figure 1: Lava removal in action

2.3.1.2 Editing Terrain

One of the primary tasks for terrain modification was to create more space for the settlement by flattening areas and removing terrain. This allowed us to include more structures and provide more functionality in each generation.

To do this, we first need to discover existing buildable areas in the selection. A "buildable area" refers to a flat and continuous piece of ground which buildings could be placed on easily. These already-existing flat areas form the basis of the eventual buildable areas. This was achieved with flood fill, which iterates through each surface block in adjacent or orthogonally adjacent blocks and records ones that have matching height values, indicating a flat surface in-game (Smith, 1979). If the area contains enough blocks with the identical height value, the selected blocks are considered a buildable area. The rest of the area will move onto the next phase of terrain editing. As this area's height level varies between blocks and is not fit to have structures placed on it, it is considered a non-buildable area at this stage.

These areas are traversed with flood fill once more. When the coordinates of each block is stored, its height and neighbouring buildable area is also recorded, allowing for further allocation and classification later. After each non-buildable area is traversed, the list of neighbouring areas is observed. If the area borders a buildable area, the area traversed will naturalise to the most bordered buildable area by adjusting the height level. This expands the existing buildable area, creating more space for structures. Conversely, if the area traversed doesn't border with any other buildable area, the list of height levels is observed and the upper and lower of the group of height levels is flattened to the next nearest height level. This generates new and extra buildable area plateaus in the peak and valley sections in the area as demonstrated in figure 2 and 3.

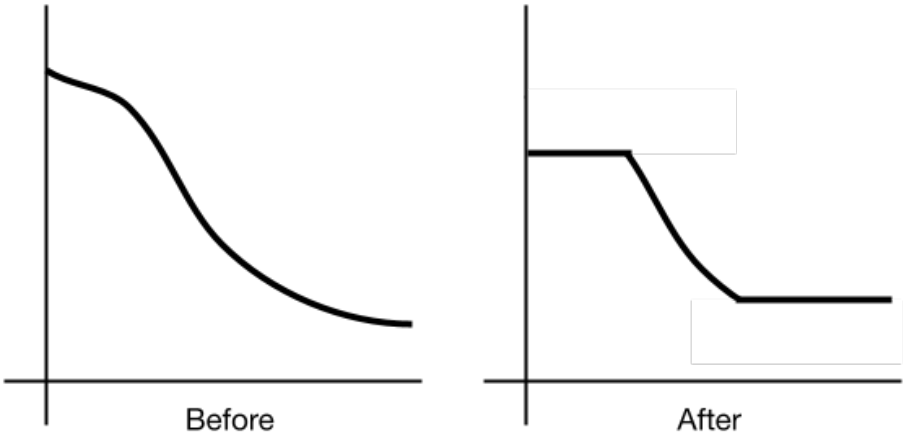

Figure 2: Side view of terrain before and after terrain modification

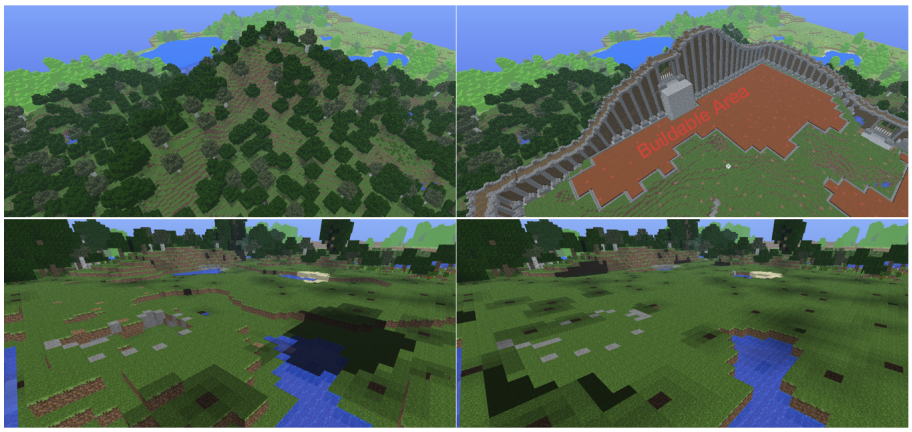

Figure 3: Before and after terrain modification in action

Besides flood fill, connected-component labeling (CCL) and Sobel operator algorithms were also considered when deciding the algorithm used for graph traversal. Sobel operator is an edge detection algorithm, meaning that an extra step would be needed to group the edges together (Kanopoulos, Vasanthavada and Baker, 1988). Both CCL and flood fill work by tranversing neighbouring cells and categorizing areas by comparing values in cells, but CCL checks only two directions (north and west of the current cell) for comparison and labels accordingly. After the first traversal, the bordering area will be merged. However, the processing time of CCL generally takes longer than flood fill for the same sized heightmap with no visible advantage, so CCL was ultimately not used (Samet and Tamminen, 1988).

2.3.2 Walls

Walls play a crucial role in our settlement, functioning as a physical barrier separating it from the outside world. They also stop hostile "mobs" (monsters in Minecraft) from entering the settlement, threatening the players and other entities inside. Given that the wall may need to transpass difficult terrains with greatly varying heights, such as mountain ranges and ravines, a workflow is required to generate the wall both functionally and logically.

To accomplish this, the walls have been made modular as shown in figure 4. Each wall section is separated into four different schematics. However, if the wall passed over water, it will also need to allow boats access in and out of the

settlement. Therefore, the algorithm will also detect if the module it's generating is above water, and place down a special water section schematic accordingly.

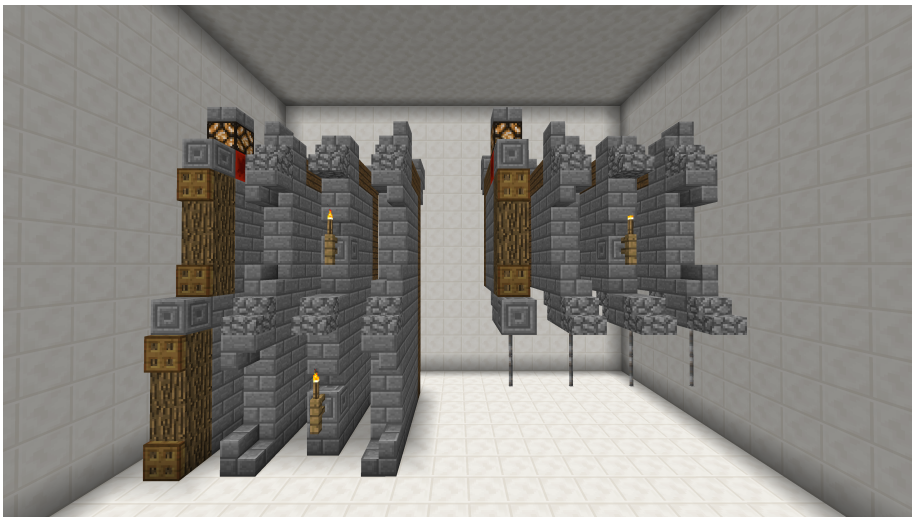

Figure 4: Modular wall sections for both normal and water sections

Under the functionality aspect of the competition it is stated that player avatars need to have access to the settlement (GDMC, 2021), so the algorithm will generate a gate on each side of the wall, in one of four different sizes, along with gate controls programmed with auto-generated commands to open/close the gate.

For all the wall sections and gates to generate seamlessly, the algorithm calculates the number of wall sections needed on each side, the position of each gate, and the gate size based on the map selection. This calculation is done for both the x and z axes.

Moving on to the inside of the wall, since the ground height level is different from the outside, the algorithm will detect this difference, place the wall base schematic based on the inner wall height, and fill in the remaining inner wall.

Originally, the walls were placed directly at ground level. However, this caused a few problems with their functionality. If the wall generates on a dip, e.g. a ravine, that section will be generated at the bottom of the ravine, creating a gap in the wall, or if the wall generates on a steep mountain, the overall flow of the walls would seem unnatural like in figure 5a. To fix these problems, we explored

several different methods to limit the block variance, and finally settled with a 10-block simple moving average. Simple moving average is a concept used in financial applications to identify price trends (Hayes, 2021), but this concept can also be used to lower the variance in wall height, calculating the unweighted mean height of the five blocks before and after the wall section. This resulted in a smoother wall like in figure 5b.

Figure 5: Before and after implementing SMA to walls generation

2.3.3 City Planning

2.3.3.1 Grid Allocation

In terms of city planning, our first objective was to determine which areas of the edited terrain were suitable to build structures on top of. We decided that a 4x4 square of blocks at the same height level would be defined as a "buildable area" in this phase, or as part of a larger one. This was accomplished by treating the

selection area as a grid composed of 4x4 squares and finding which of them were buildable. Additionally, we iterated through all 16 unique possibilities of starting x and z coordinates for the grid in order to find the one that resulted in the most buildable areas, to maximize the amount of structures we would be able to place in our settlement.

We later decided that each buildable area should be surrounded by a border, which would be useful for the eventual generation of city paths connecting all areas together. This required further changes to the grid system, checking that each buildable area also had space for a border around the outside of it, in some cases modifying the terrain slightly to make this possible. This was accomplished by first modifying the initial grid calculation to track if the 6x6 area surrounding each 4x4 area was also of the same height level, then implementing further methods allowing a 4x4 area to "claim" borders adjacent to it by editing their height level to be the same. Only buildable 4x4s that had been proven to have the most adjacent buildable 4x4s of any area surrounding them were allowed to claim a border in this way, to ensure that the extra space went to the largest buildable area, which would then be able to fit more and larger structures.

After this had been determined, the code for the border itself was added. Composed of stone brick stairs, it connects seamlessly to both the stone brick floors used for each area of the village and the paths connecting them. Calculations are present in the code to ensure that the stairs used in each border are facing in the appropriate direction depending on which side of the border they are part of.

At the end of this process, a 2D array is produced which contains a 1-to-1 mapping of the x and z coordinates of the selection area. Each block is represented by a number; 0 indicates a block that is not part of a buildable area, 1 through 4 indicate the four stone slab entrances inside each of the four gates, and 5 and up indicate each separate buildable area present in the rest of the selection, including their borders. This array provides a representation of all buildable areas in the settlement with a unique identifier for each one. The purpose of this is to make it easier for the areas to be identified and connected together with paths.

2.3.3.2 Structure Allocation

With the numbered 2D array of buildable areas, it is possible to start assigning structures to those areas. First of all, all of the available locations that fit each of the building shapes are discovered, including right angle rotated variants of the building shapes, allowing structures to be placed in different orientations. Then, the algorithm will attempt to assign structures to locations with the location discovered alongside the weighting and count of each structure type. The allocation of buildings is completely random, including the location, orientation, type, and variation of the building, ensuring each generated settlement is unique.

However, each structure type has a maximum number and weighting in place to constrain how often it can occur in each settlement. For example, we decided the town hall should be placed in the settlement if the terrain allows for it, and that only one instance should exist in a single settlement.

With the list of weighting and max occurrences, prioritisation of different structure types is possible, allowing similarity and variability between settlements and randomness of building. The allocation process will execute ten times to try to obtain the most optimised settlement to generate. A fitness score is calculated for every sample generated by calculating the type of structures assigned and the total area occupied by structures. A well-developed settlement should include a good amount of functional structures and attempt to optimise the area available, so the fitness score takes account of the building types and occupied area. It acts as a merit index to compare the potential settlements to generate. At the end of the process, the allocation set with the max weighting in the generation is chosen to be used for the final settlement.

2.3.3.3 Tree Formation

Next, we needed to consider adding lighting throughout the settlement as hostile mobs would spawn in areas that have a light level lower than eight (Minecraft101, 2011). Therefore, we created trees that incorporated lighting to be placed in the unbuildable areas within the settlement. This fulfilled both functional and aesthetic requirements as the trees would provide safety for villagers and look more visually pleasing, as we removed all trees in the selection at the beginning of our algorithm.

To accomplish this, we looked into circle/rectangular packing as we needed to make sure all unbuildable areas would have a sufficient light level in the most efficient way. However, while researching circle packing it became clear that it would not suit this particular problem because the concept does not allow overlapping (Weisstein, 2021). This meant that there would be small areas without sufficient lighting, which would allow mobs to spawn. As a result, we had to create an algorithm ourselves.

We looked at each tree design and found that all had some form of lighting three blocks above the ground and lighting around the trunk so this acted as the centre of the tree, as shown in figure 6. We then created an area around each 'tree' and looked at which point the light level falls below eight in which somewhat of a grid was formed. The distance between each tree from left to right would be (11, 1).

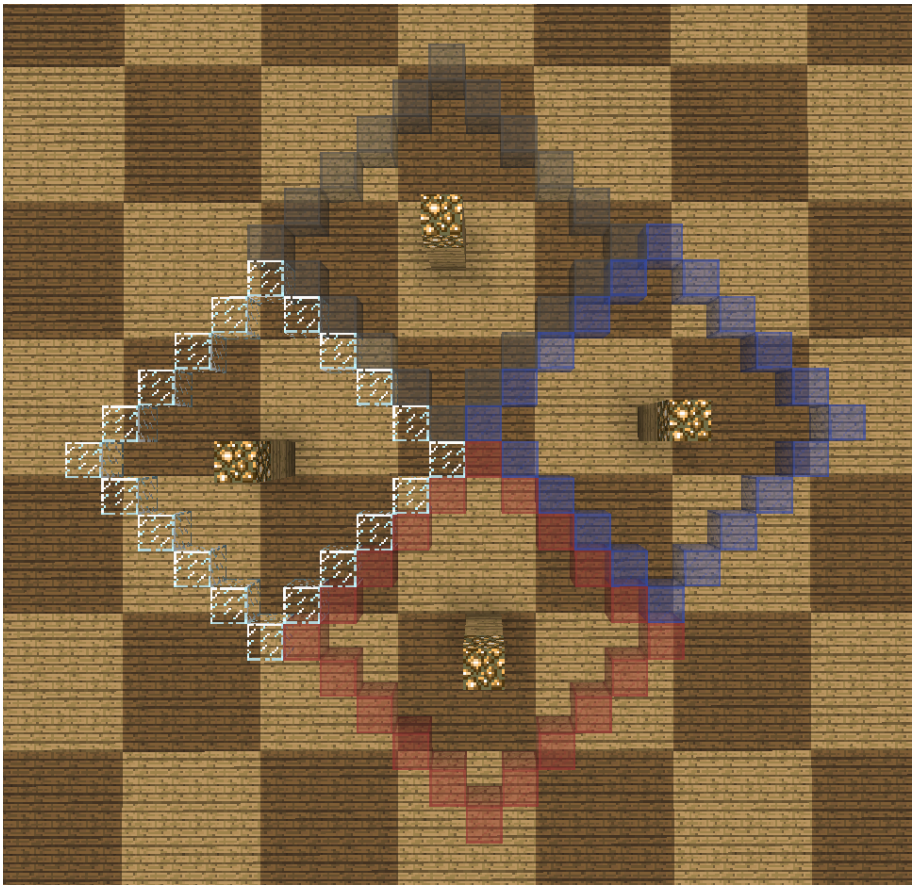

Figure 6: Visualisation of how trees will be organised

With this in mind we had to think about where to spawn the trees. We had to check the 3x3 area to see if any of the blocks were buildable areas, a gate or the entrance. If so, it would remove it from the array and not spawn a tree there.

Once they were generating correctly, we evaluated the look of the trees and found that they looked too uniform, as seen in figure 7a. Thus, a one block variance was implemented to make the trees look more natural, shown in figure 7b. To do this, we checked the area around the tree's position and randomly chose a new position for the tree to generate, which would then replace the old position. This resulted in more trees being placed, as it checks other positions to place down the tree and continue the algorithm.

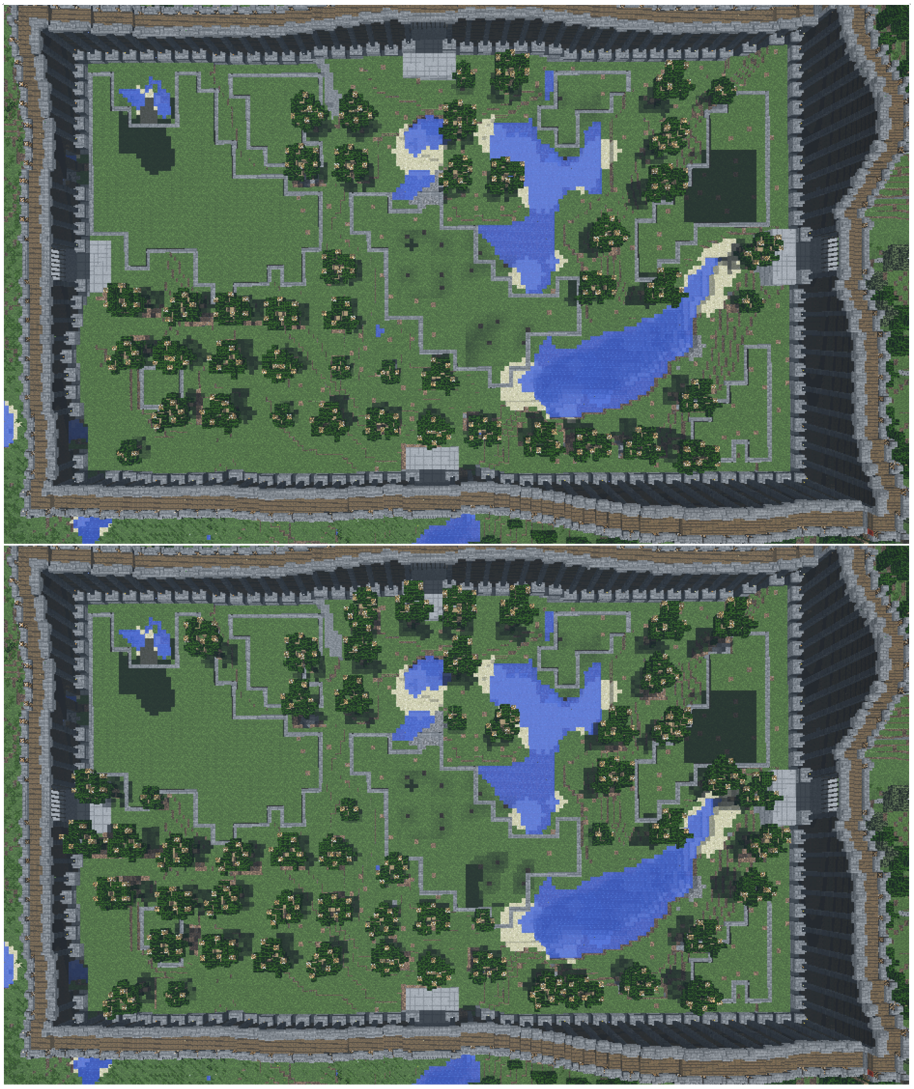

Figure 7: Before and after randomised one block variation

2.3.3.4 Path Finding

Path finding is essential to the settlement, as it allows for the creation of a path connecting all buildable areas and gates together, providing both functionality and aesthetics. The path finding algorithm for our settlement is separated into four main parts, finding all possible combination of minimum distances between each area, running Prim's minimum spanning tree (MST) algorithm to find the shortest paths to link all areas, followed by the A* algorithm to find all shortest accessible

paths, and finally placing those paths down.

In order to find the minimum distance between each area, the algorithm calculates the shortest Euclidean distance of all possible combinations for each area. A test case was run for the path finding algorithm, and represented graphically in figure 8 9 and 10. Figure 8 shows a representation of all possible paths linking to area 0. This process is then repeated for all other areas.

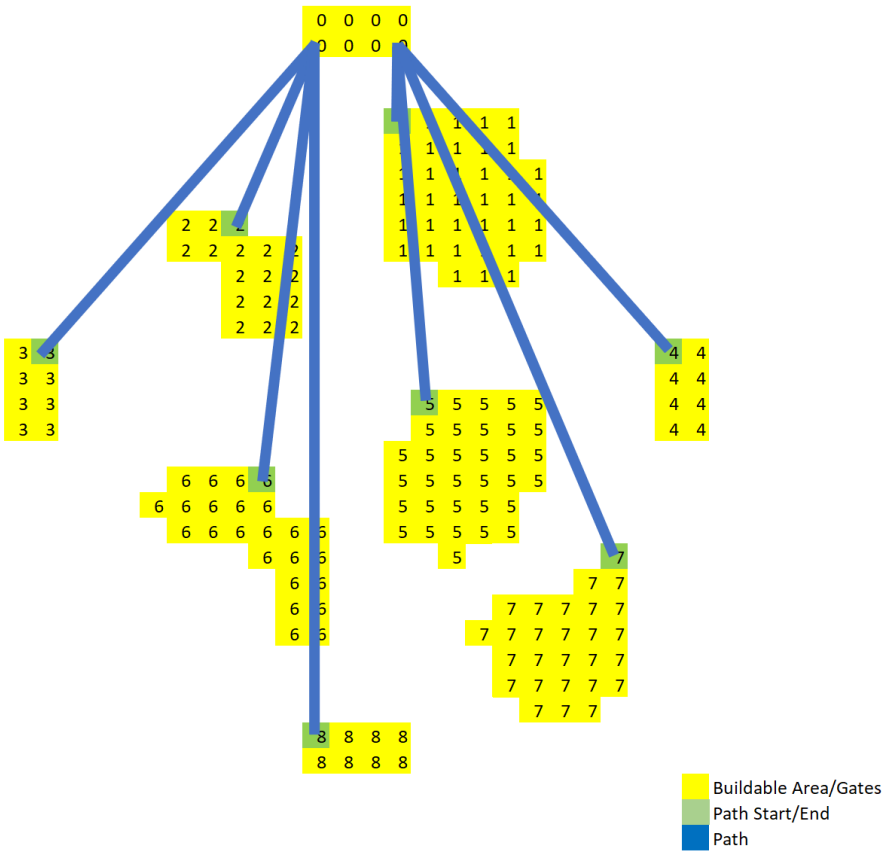

Figure 8: Minimum distance to link area 0 to all other areas

In order to find the shortest distance linking all areas together, we decided to use a greedy algorithm, Prim's minimum spanning tree (MST). MST finds edges with minimum cost in the graph that connects a reached node to an unreached node; this process is repeated until all nodes have been reached (Prim, 1957). This concept is then used in the algorithm to select paths with the minimum distance

that links all areas together in the test case, as shown in figure 9.

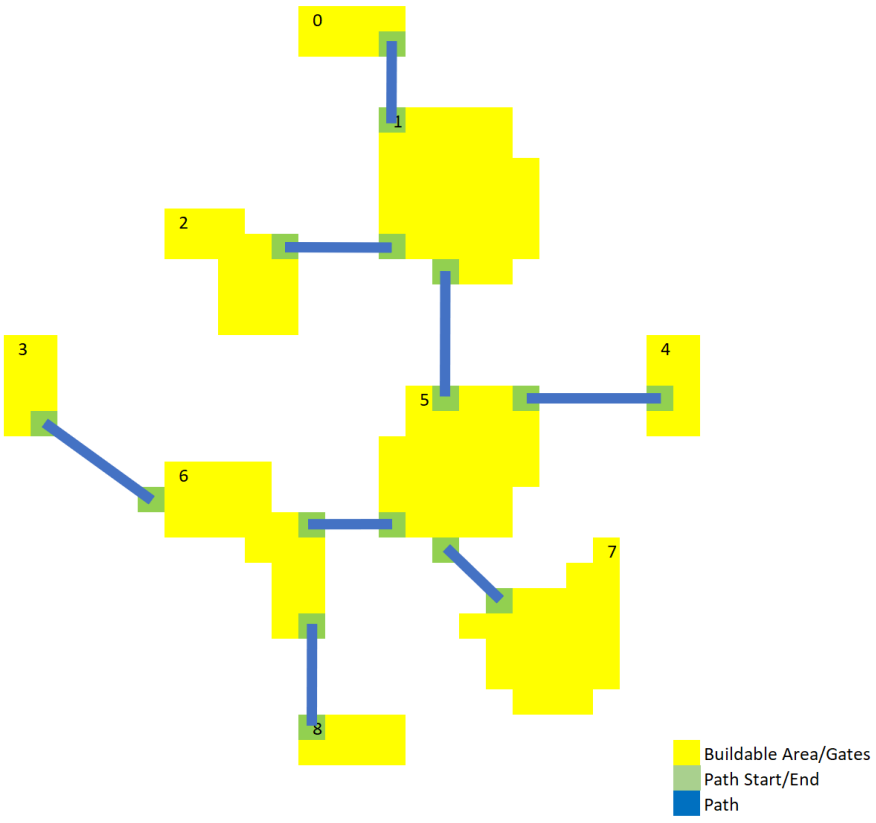

Figure 9: Prim's algorithm to find the minimum spanning tree

Dijkstra's algorithm finds the shortest path between two nodes. Its concept is very similar to Prim's MST; it finds edges with minimum cost in the graph that connect a reached node to an unreached node, then traces back its parents to form a path linking the start and end node (Dijkstra et al., 1959). However, Dijkstra's algorithm is not the most optimised for our settlement generation as it looks up all possible routes from the starting node. A* algorithm on the other hand uses the same concept as Dijkstra's, and optimises it by using a heuristic function, selecting nodes that seem to provide a better locally optimal solution (Hart, Nilsson and Raphael, 1968). Due to this, our algorithm will then run A* algorithm to find the shortest available path, pathing around trees and other obstacles. In figure 10, the 1-2 path and 5-7 path had to reroute around the obstacle.

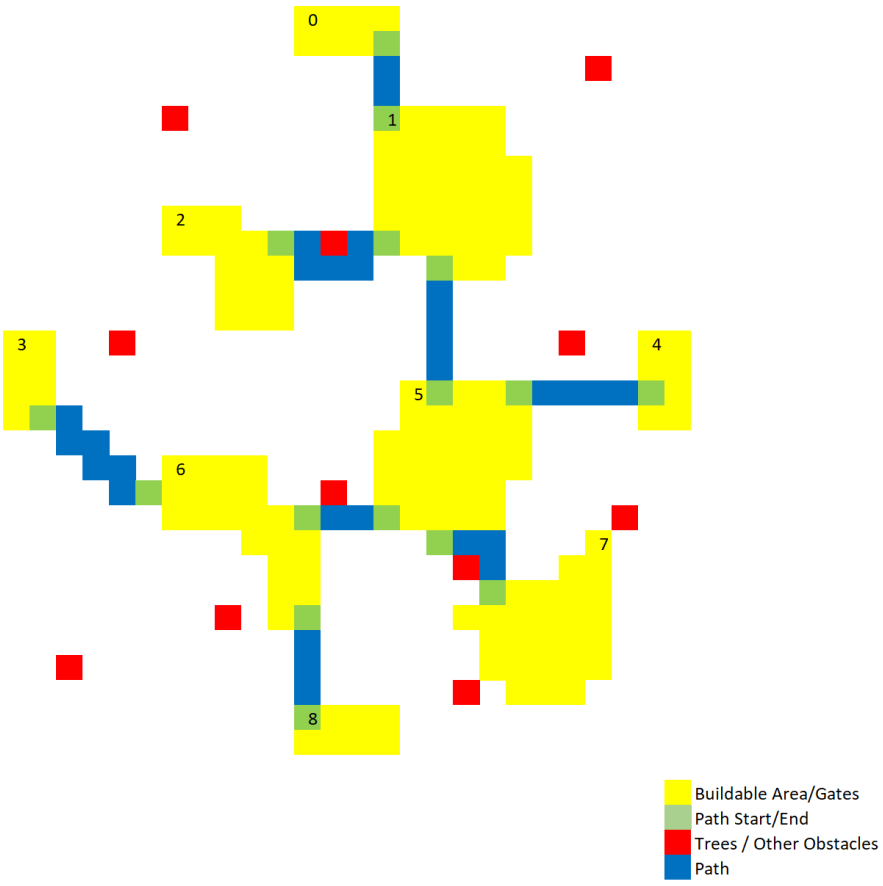

Figure 10: A* path finding algorithm

The final step of the algorithm is to place down all the paths and fix remaining details, such as placing ladders if the height difference is more than one block (figure 11), or removing one block above the path if generating through farmland, etc.

Figure 11: Ladder generation when block increase is more than 1

Overall, the path generation is able to create an accessible path that links all buildable areas and gates together, like in figure 12.

Figure 12: Path generation in Minecraft

3 Discussion

3.1 Results

Our algorithm produces a realistic settlement that considers both Minecraft and real-life problems. When executed it is able to smoothen and modify the terrain in a way that is natural while creating spaces for settlement. Then, it plans and generates structures such as walls, housing, and other structures that occur in ordinary human settlements. These also include farms, a town hall, housing variations, libraries, as well as paths that connect different sections of the generation. With computational randomness and pre-made structures, the algorithm leverages variability from structure allocation and constancy from structure type. Compared to the game's default settlement generation, our algorithm provides a greater range of structures, better pathing, workflow generation, and projection, additionally resolving the problem of elevated and inaccessible structures. If compared with past entries, we believe that our solution would place relatively highly, as it is able to fulfill most of the aims in the four criteria listed by the host. Although there are still areas that can be enhanced with further development and additional functionality that can be added, we think that our achievement is largely successful.

Figure 13: Top down view of settlement in MCEdit

Figure 14: Top down view of settlement in Minecraft

3.2 Evaluation

3.2.1 Feedback

During the presentation of our project, we attracted a lot of attention and interest from visitors curious about our process. Overall, we received positive feedback, but in order to further evaluate our project, we created an online form for visitors to give their honest opinion of the settlement. The questions were geared toward the criteria of the competition, allowing us to self-assess in a similar way. Currently, from that form we received several responses, in which for adaptability, aesthetics, and narrative 100% of participants ranked our settlement 4 or 5 out of 5. For functionality, we received somewhat lower scores as at the time we did not have paths or farmland. This led us to prioritise those features in our product backlog for the remaining sprints.

Responses → Criteria ↓	1	2	3	4	5	6	7	8	Average	Variance
Adaptability	4	4	4	5	5	4	5	5	4.5	0.29
Functionality	4	4	5	5	3	4	4	4	4.125	0.41
Narrative	4	4	4	5	5	5	4	4	4.375	0.27
Aesthetics	4	5	5	4	5	5	5	5	4.75	0.21
Average	4	4.25	4.5	4.75	4.5	4.5	4.5	4.5		

Table 1: Feedback questionnaire score

3.2.2 Challenges

During this project, we faced a multitude of different technical and non-technical problems. One of the most challenging issues we faced was during research, as we came across a plethora of different methods to solve problems. As a result, this meant that we had to conduct research into each topic. In each case, there was extensive research involved where some would be computationally challenging and had insufficient resources to back up proof of concept. This meant that a lot of algorithms we looked into were scrapped, which made our progress in completing tasks feel slow. While this enabled us to build foundations, develop and learn better approaches for our algorithms, it was rather time-consuming to find the best way to solve our problems.

Furthermore, accounting for the variable terrains in Minecraft was also frustrating. Components may react unexpectedly for certain terrain and because of the random world generation in Minecraft, it is difficult to encapsulate all these conditions in our code. Hence, extensive time was spent on performing tests on an assortment of terrains.

Adapting to some of the outdated components was also difficult for us; as MCEdit is no longer supported, we have experienced extensive issues with instability during development, prolonging our time spent on testing and planning.

3.2.3 Alternative Approaches

3.2.3.1 Structure Allocation

In the allocation process, we adapted elements like fitness and generation from genetic algorithms, because the allocation method was first developed with the vision of constructing an evolutionary algorithm, where the generation will improve as the sample progresses through the algorithm. However, this development was halted as we encountered the problem of packing different rectangles in a rectilinear polygon.

A rectilinear polygon is a polygon formed only with edges intersected at right angles. This is a NP-Complete problem, meaning it is difficult to come up with a definite solution. One workaround is to obtain the approximated solution of the problem instead. One approach could have been to partition the rectilinear polygon into rectangles in advance, reducing the complexity of the calculation. However, pre-partitioning may reduce the usable area and limit available structures. This leads us to the option of adapting a genetic algorithm, where a solution is developed and attempts to improve as processes like mutation and crossover take place in later generations. We then realised that the algorithm was unable to assign structures to its optimal location, as the randomly selected location was unable to cater the randomly selected structure and dropped the allocation, causing multiple generations filled with the minimum sized structure. This inspired the decision to discover locations for each structure type in advance, ensuring the structure and the location randomly selected will always be allotted.

At the same time, we ran into difficulties carrying out mutation and crossover to evolve the existing samples. Mutation is not possible because the allocation of structures is interconnected, meaning if one of the structures is replaced, the rest of the structures need to be replaced as well, as the area involved will not fit a structure that enhances the fitness. Additionally, crossover is not particularly viable, as the operation may slice up an assigned structure, resulting in a false structure, requiring the entire allocation to be recalculated and reallocated. This caused us to standardise the sizes of structures and buildable areas into clusters of 4x4 blocks in-game. This constrains the flexibility of allocation in favor of reducing the ambiguity of the calculation.

Because of this, the allocation method currently adopts standardised size of

structure and slots and allocates with pre-calculated slots, speeding up the performance time. In the future, it could be possible to reintroduce genetic algorithms to perform the allocation, as each of the buildable areas in a sample is standardised and identical across the population. Figure 15 demonstrated how structures are allotted in a buildable area within the selected area. To perform crossover, exchange a selection of buildable areas, as shown in figure 16. For mutation, reallocate a selection of buildable areas as shown in figure 17. Although the processing time will increase in exchange for performing mutation and crossover, the reintroduction of genetic algorithms would enhance the optimisation of the buildable area and lead to a more concise settlement.

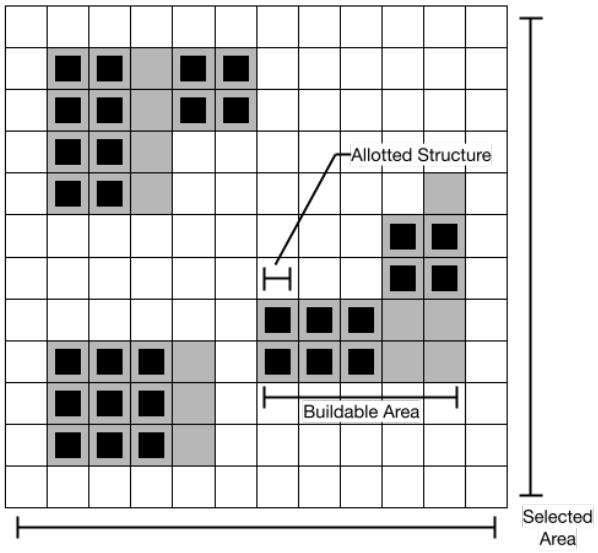

Figure 15: Definition and relationship of area in allocation

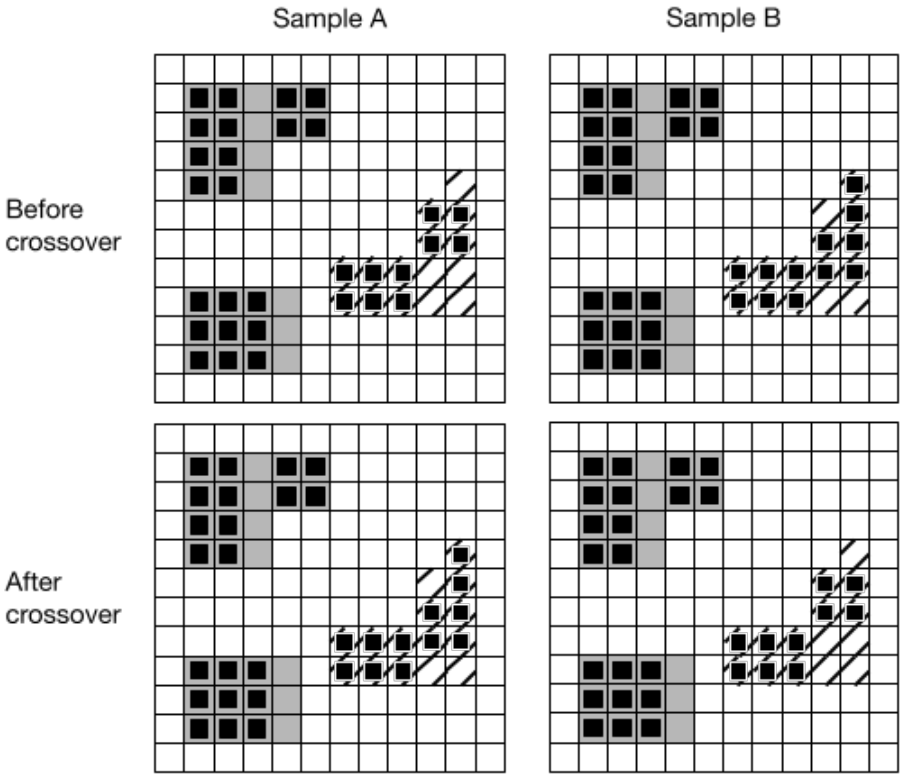

Figure 16: Crossover in action by exchanging buildable area

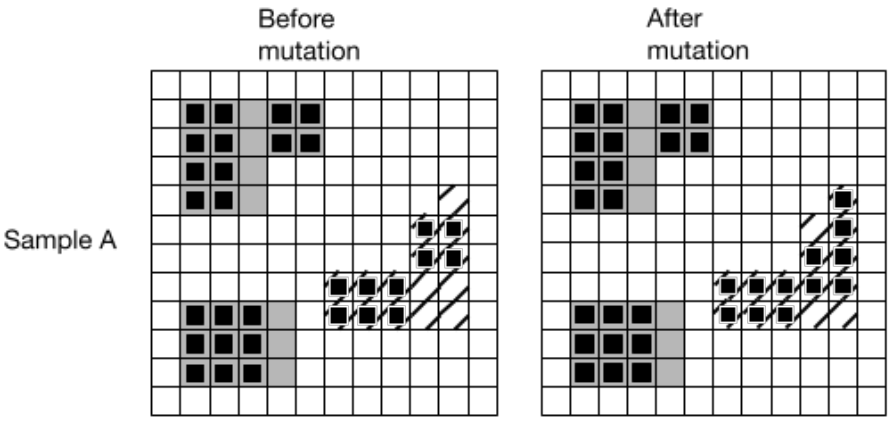

Figure 17: Mutation in action by recalculates a buildable area

3.2.4 Further Development

Due to factors such as time constraints and task prioritising, we have reduced or abandoned several features we had originally planned for. On reflection, these plans may help with generating a more realistic and complex settlement in the future. One such idea was modular building structure, in which the same structure type may be formed by sections or interiors with different designs, or structures with expandable sizes. One example of this that was implemented was the village walls, which were formed by using different schematics while reacting to the terrain and size of the generation. However, the plan to use this for other structures was dropped, as the development time would have been too long.

Another feature we considered was biome modification. This would have involved building design/materials that would reflect the biome they were located in, allowing for biome-specific structures. The aim was to reflect the unique setting of the biome and increase the diversity of the structures, giving our settlement a stronger narrative. This plan was also dropped due to time constraints, with other tasks being prioritised over it.

Other plans that were discontinued include several additional structure generations, a faster mode of transport through the settlement, and integration with machine learning in allocation. These features would improve a player's quality of experience, but would have been too time-consuming to develop given the scope of the project.

4 Conclusion

Over the course of the project, we have both learned a number of new technical and non-technical skills, and improved on existing ones. We developed not only as individuals, but also as a group by adapting our ways of working and communicating in order to overcome problems. Technical skills we developed include learning Python programming and understanding the workings of several algorithms. As for non-technical advancements, our most significant improvement involved better managing tasks across multiple team members over a long period of time, and learning to compromise and make hard decisions on features to focus on given the limited amount of time available to us. As these skills would be beneficial when

working on time-sensitive projects as part of a team, we believe that working on this project has made us better prepared for our future careers.

5 Epilogue

5.1 Submission

As this project is based on the GDMC rules and guidelines, we have decided to enter the competition with this project. This is a valuable opportunity for us to gain constructive feedback from the professionals and the wider community. From this, we want to reflect on our progression and achievements by allowing us to compare ourselves with other entries and learn from them.

Acknowledgement

We would like to thank Dr. Anna Jordanous for supervising the project and providing consultation and guidance throughout. Furthermore, we would like to thank all those who participated in our online form and provided us with valuable feedback. Special thanks to Peanut Chu for invaluable emotional support.

References

Dijkstra, E. W. et al. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1), pp. 269–271.

GDMC (2018). Generative design in minecraft. *Tandon School of Engineering, New York University http://gendesignmcengineeringnyued.*

GDMC (2021). Settlement generation challenge 2021. *The GDMC Competition https://gendesignmcwikidotcom/wiki:settlement-generation-competition.*

Hart, P. E., Nilsson, N. J. and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2), pp. 100–107.

Hayes, A. (2021). Simple moving average (sma). *Investopedia https://wwwinvestopediacom/terms/s/smaasp.*

Kanopoulos, N., Vasanthavada, N. and Baker, R. L. (1988). Design of an image edge detection filter using the sobel operator. *IEEE Journal of Solid-State Circuits*, 23(2), pp. 358–367.

Minecraft101 (2011). Lighting. *Minecraft101 http://wwwminecraft101net/t/lightinghtml.*

Prim, R. C. (1957). Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6), pp. 1389–1401.

Salge, C., Green, M. C., Canaan, R. and Togelius, J. (2018). Generative design in minecraft (GDMC): Settlement generation competition. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, New York, NY, USA: Association for Computing Machinery, FDG '18.

Salge, C., Green, M. C., Canaan, R., Skwarski, F., Fritsch, R., Brightmoore, A., Ye, S., Cao, C. and Togelius, J. (2020). The ai settlement generation challenge in minecraft. *KI-Künstliche Intelligenz*, 34(1), pp. 19–31.

Samet, H. and Tamminen, M. (1988). Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(4), pp. 579–586.

Smith, A. R. (1979). Tint fill. In *Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA: Association for Computing Machinery, SIGGRAPH '79, p. 276–283.

Weisstein, E. W. (2021). Circle packing. *MathWorld https://mathworldwolframcom/CirclePackinghtml*.