

Procedural Map Synthesis and Generation

Abstract

Utilizing various procedures that are normally used to generate 2D images and genetic algorithms, I explore the feasibility and effectiveness of applying these techniques to maps or fully three dimensional images. The maps that this system creates are rendered in Minecraft and MCEdit, a Minecraft map editor, since this easily allows 2D image techniques to 3D maps. Minecraft maps consist entirely of voxels, a pixel represented in three dimensions rather than two, essentially making a Minecraft map analogous to multiple images layered on top of each other. Unfortunately, due to added dimension of height, a huge problem of scale results. Despite having variable width and length, all Minecraft maps have a fixed height of 256 voxels or blocks. Even the minimum allowed map size, which consists of a width and length of 256, is 65536 voxels per map or 256 times larger than an image of the same width and height.

1 Introduction

Virtually every 3D video game features at least one map of some kind but the granularity and dimensions of these maps varies widely. This leads to a problem when you have games that must have maps that grow or exceed the capacity of the developer to create them. Keep in mind resource creation is generally expensive and to create maps for games like Mojang's Minecraft or Hello Games's No Man's Sky, which has 2^{64} planets, by hand is practically impossible even with unlimited funds. Therefore, these games utilize a technique called procedural generation in order to get around the problem of very large maps. In procedural generation various, rules are put in place that define what kind of blocks are generated as the player typically through structures containing a specific order of blocks, relations between blocks, and a seed that programmatically determines the entire map.

In this project, I attempt to use methods used to create or grow 2D images to as new methods of procedural generation. Rather than an integer seed, a source map is used that determines how new maps are created. The seed in most procedural generation systems will always create the same map [6]. These systems rely on vast scale to present novelty to the player and randomizing the seeds themselves. In this project, unique maps are created with the same seed essentially, but this leads to an efficacy question.



Figure 1: A No Man's Sky planetary map. Although it appears to be contiguous and detailed, it is actually discrete where every structure is a predefined asset. Since it is discrete, it generally easily for repetition to occur.

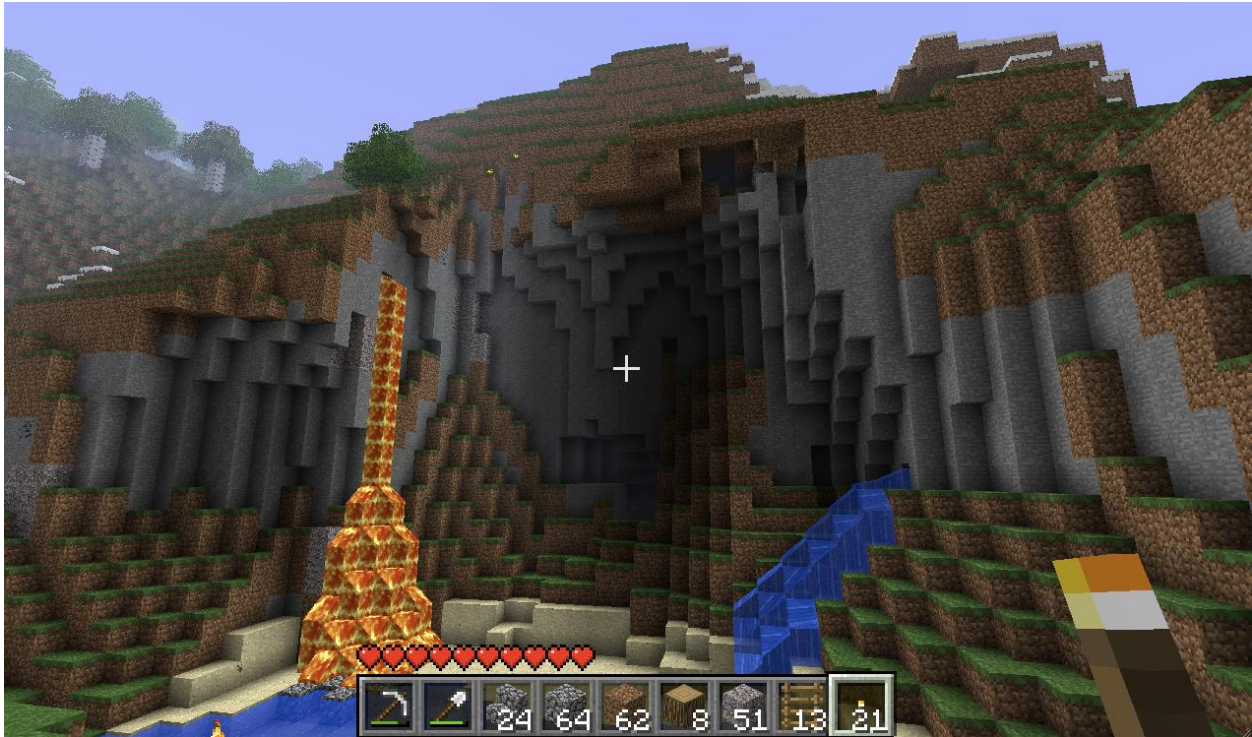


Figure 2: A Minecraft map. Every structure is defined as a series of blocks or voxels which saves the developer the cost of creating unique assets. Minecraft uses 3D voxels in the same way that pixels are used in 2D images to create a map that resemble natural structures.

2 Map Format

In Minecraft, maps are generated in chunks as the player is a draws close to its edges. Chunks are the smallest entity that the algorithm generates and saves to the player's hard disk. Every chunk is 16 by 16 voxels with a height of 256 voxels leading to a total of 65536 voxels or blocks. In the most recent version, these chunks are stored in region files, which contain a maximum of 1024 chunks, information about the types of chunks contained in the region file also known as the biome, and all the game characters within that region. For instance, if a region consists of an ocean biome then its blocks will consist of mainly water, air, and small pockets of sands that represent islands.

For the purposes of this project though, the information contained in the region files are extraneous and only utilized to enhance the game play experience of Minecraft. Unfortunately,

the priority of gameplay in Minecraft influences the save structure such that accessing the map data is an annoyance. It is simply not built to normally allow the creation of save files generated from a system outside of Minecraft itself. The only way to create a map that Minecraft will render is via a region file, but because a region file can vary in both chunk amount and distribution it becomes a hassle to organize. Also keep in mind that while a region file can hold up to 1024 chunks, whether or not it holds a chunk will depend completely on its position on the map. In a typical Minecraft save folder, there are multiple region files each containing portions of the entire map as a whole.

Therefore, it was necessary to find a new format that was either supported by Minecraft or easily converted into Minecraft save file. I found this in the .schematic file type and Minecraft map editor MCEdit.



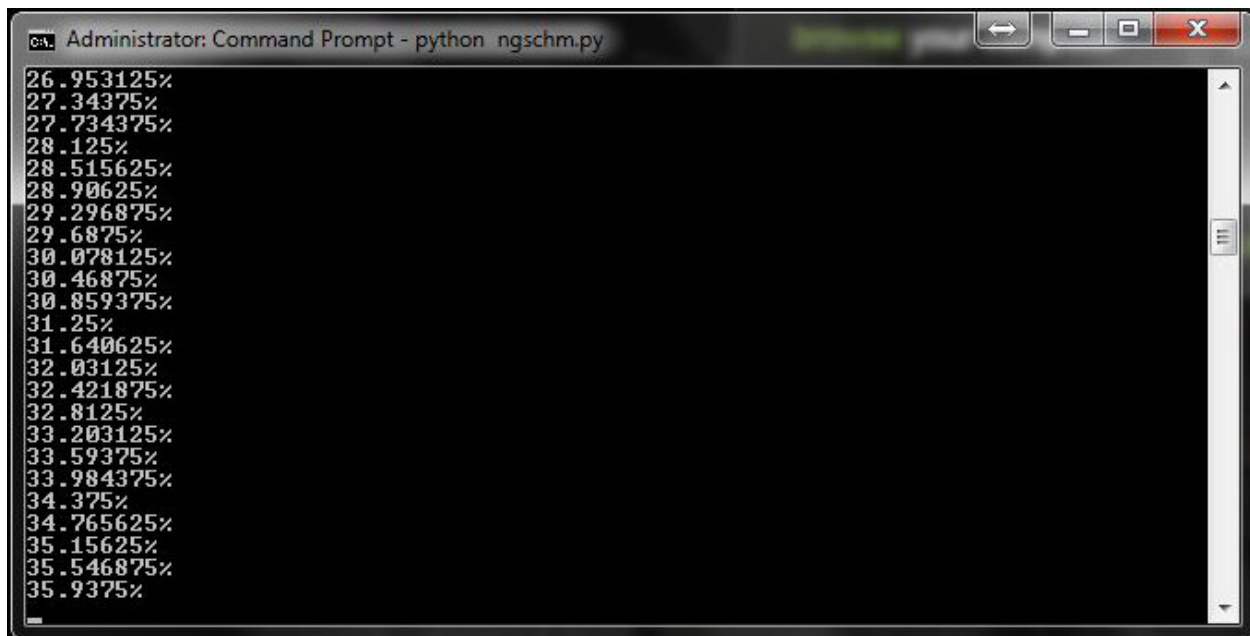
Figure 3: An artificially created map saved as a .schematic file and rendered in MCEdit. It can represent maps as two dimensional chunks (left) or as fully three dimensional models. The .schematic file is versatile, easy to work with, and can promptly be inserted into a Minecraft Map for further testing.

The .schematic file type was created to store the schematics and plans of artistic construction in Minecraft's creative mode that can be shared and easily modified. Whereas the game normal save files prioritize optimization and game mechanics, .schematic files are generally unconcerned with region file format order and will instead opt to store everything as huge arrays. While this has a problem of scale, it makes it really easy to access the map array which is literally just an array of block types. In essence, this is the digital form of the map before it is rendered and represented by blocks. There are a total of 195 total block ids that corresponds to different kinds of blocks such as grass or lava blocks. These are represented in the array as unsigned bytes. Unlike region files and the chunks therein, a .schematic file is trivially easy to create and an artificial map of my creation can be packaged in it.

3 System Setup

In the absence of a seed to algorithmically generate a map I create a sample map generated by Minecraft in order to create my own maps. An efficient sample must be larger than the maps I plan to create, and it must contain a fair distribution of blocks across the spectrum. It is important to keep in mind that the block types 0-18 are the most common, with blocks 0 and 1, air and stone respectively, making up the majority of the map. If for instance, the sample contains a high proportion of water blocks, every resulting map will be predominated by water blocks.

This sample map referred to in the project as the source which is stored in a three dimensional byte array. In Python, the Numpy library has native support for N dimensional arrays of one type, so this is also a simple matter as well. Unfortunately, Python was not built for scientific computation and even though Numpy alleviates this to some degree, many of the methods that are examined in this project are entirely too slow. There are a few ways to optimize this problems further such as multithreading changing the method by which a map is grown, but this will be covered in further sections.



```
Administrator: Command Prompt - python ngschm.py
26.953125%
27.34375%
27.734375%
28.125%
28.515625%
28.90625%
29.296875%
29.6875%
30.078125%
30.46875%
30.859375%
31.25%
31.640625%
32.03125%
32.421875%
32.8125%
33.203125%
33.59375%
33.984375%
34.375%
34.765625%
35.15625%
35.546875%
35.9375%
```

Figure 4: One of the issues of using Python. Using a variant of Efros and Freeman's Image Quilting technique on an 8 by 8 chunk map, the program only completes 36% of the map after approximately 3.5 days of runtime [1].

4 Genetic Algorithm

The initial strategy was to randomly generate the initial population by simply generating an array of maps, themselves each being a three dimensional array of random integers. This primary population is graded via a fitness function that gives a numerical score of how well-founded a map is. The maps are then sorted by this score in descending order, the top few of which are included in the next generation along with a few lower scored individuals to ensure diversity. Furthermore, an even smaller portion of this new population are mutated to increase the chance of variance further. Finally, more individuals are added to the population by combining the existing individuals referred to as the parents, to create children such that the population size is consistent every generation. By using Numpy instead of Python's native list structure, the program was sped up significantly from 30 minutes on `gschem.py` to 7 minutes on `ngschm.py`, the Numpy variant.

For the fitness function, I used the Counter function, a tool from the Python Collections library, which returns a dictionary containing the occurrences of every element in the array. When run on map, it reports the occurrence every block type in said map, and while it adds some overhead, it is the most efficient way to determine the “character” of a map automatically. Stored on disk is a file of the dictionary generated when the Counter function was used on the source, divided by the size that all the artificial maps were set to, 64 chunks or 4194304 blocks. Every map created by the program had this standard size to minimize discrepancies as well as be large enough to display noticeable patterns. Saving the file on disk, also alleviated the problem of having to run the Counter function on the source file, which would take a couple of seconds, every time the fitness function would be called. With the default settings of a population size of 20 for 10 to 100 generations, even running the Counter function once on the smaller map was costly. After this analysis was complete, the differences in block amount and proportions were measured for each block type between the source and the individual, the absolute value of these differences were summed together, and the score is then subtracted from 1 million. With Numpy this fitness function nearly doubled the run time of the genetic algorithm (fig 6) from 7 minutes to 13 minutes, but this was still much shorter than the native Python implementation.

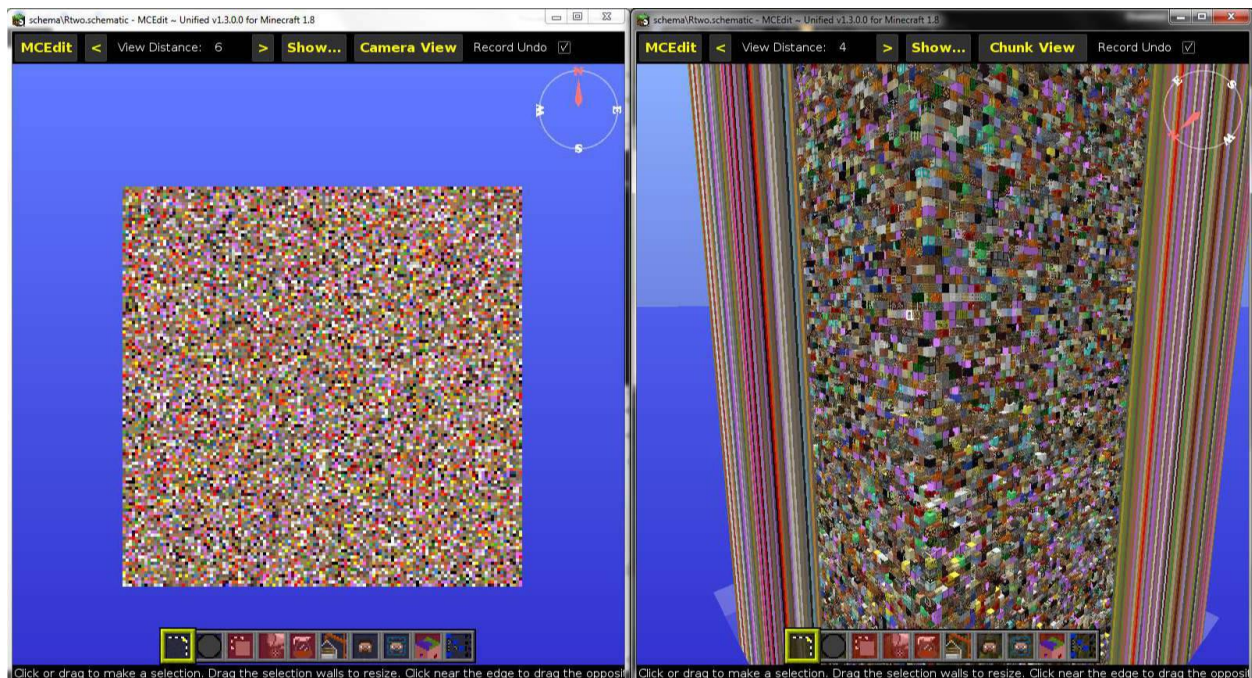


Figure 5: Results of randomly generated individuals with a genetic algorithm of 100 generations applied.

Despite the speed of the program, the results were dismal. Early products showed no discernable pattern, and resembled nothing like the maps Minecraft itself generated let alone being completely unplayable. The problem is simply one of scale again. Since a map consists of 4194304 blocks each one being any one of 197 block types, this leads to 4.613×10^{1304} possible maps. Only a small subset of these maps would resemble the source. To make matters worse, even with 100 generations of 20, an program that took about 13 minutes to run to completion, would only cover 20000 of these possibilities, not even scratching the surface.

5 Quilting

In order to circumvent the issue I decide to use a modified form of Efros and Freeman's image quilting technique [1]. In their image quilting technique, Efros and Freeman also rely on a source image for generating a new image. Unfortunately, there are a couple of serious issues that prevent simply applying the algorithm to three dimensional maps.

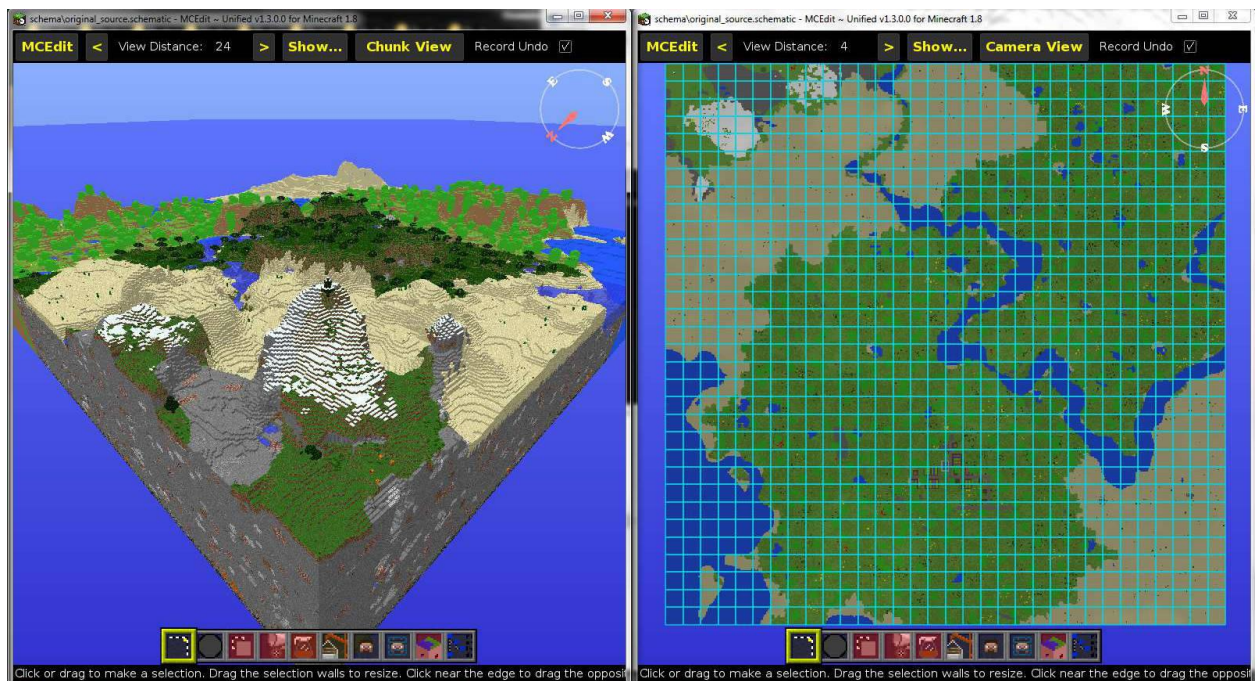


Figure 6: The fully rendered source map of 512 by 512 by 256 blocks (Left). The source map with chunk boundaries highlighted (Right). While the maps created by the project are 8 by 8 chunks or 4194304 blocks, the source map is 32 by 32 chunks or 67108864 blocks.

The algorithm relies on finding an element in the source and then using that element's neighbors to decide how the artificial map will grow. In Python, searching an array or list for element, will always return the first instance of a block type. This would mean that the algorithm would simply create the same map repeatedly since the relationship between blocks would be static. Simply randomizing the source so that the distribution is more normal is a naïve solution. Unfortunately, this operation is very slow, 0.4 seconds per block resulting in 311 days to simply finish one individual. Clearly, this is not feasible.

To combat this issue, I organize the source map into a hash table where every block type is an array of indices of the blocks directly above or to the right of this block type. The reason I limit the included blocks to those in the north and east directions is because these are the directions I grow the artificial map. Each block type corresponds to the index of a subarray that contains the index of every occurrence of this block type in the source. While much faster than the first attempt, this method is also too slow to be useful. In figure 8.1, timeit reports that the function on 1000 iterations takes approximately 6 seconds. In the best case on a full map, this should take an estimate of 6 hours to complete for one individual. However, Figure 4 shows that in reality this algorithm takes much longer to complete, only completing 36% in 3.5 days.

The last strategy was to use the source map to build a simpler hash table. Rather than each block type containing an array of occurrences, they simply contain an integer that represents the probability of occurrence, which can be counted using the Counter function. Using this a map can be grown by calculating the most likely neighbor of each. In order to create 20 individuals using this method took approximately 10 minutes as seen in figure 8.5. Combined with the genetic algorithm, it takes about 1.23 hours to complete [Fig 8.3]. However, since the

quilting technique generates a novel map initially, the addition of the genetic algorithm adds unnecessary cost since it only shuffles the maps rather than create a variety of new ones.

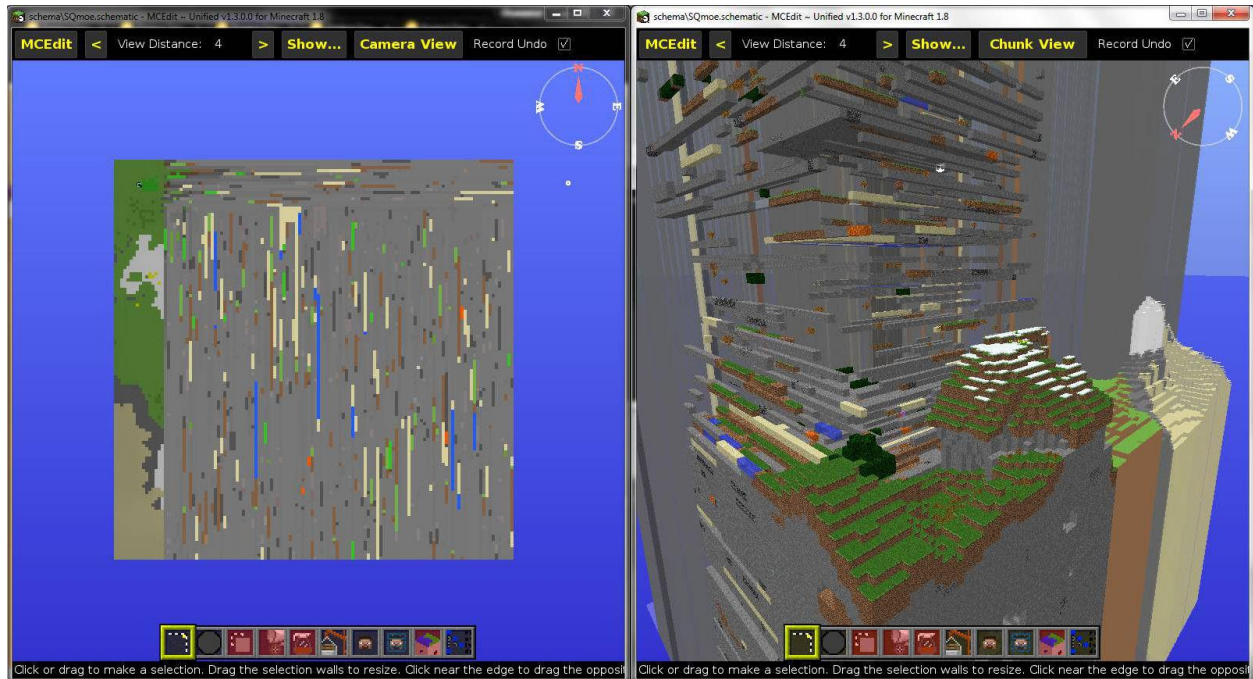


Figure 7: The result of the stable quilting mechanism a random selection of the source is chosen, and from that the map is grown using the probabilities found in the hash table.

While this quilting method is useful, it has problems in creating a well-founded map. As shown in figure 6 and figure 9, after a height of 62-64 blocks the most common block type is block type 0 which corresponds with Air, while under that height block type 1 or stone is the most common. In figure 7, it is clear that while this rule holds true for the map underneath the height limit of 62, referred to in game as the “sea level”[9], it does not hold true above this altitude limit. Furthermore, the quilting technique works better when working on small samples at a time rather than one big map. As shown in figure 7 once more, the patterns of tiles become more uniform as the maps growth progressing suggesting a convergence to the most common block type. This is problematic because it decreases the probability of certain blocks occurring which is makes rare blocks even rarer and common blocks even more abundant. This is why it appears to be dominated by stone. The solution to these problem is simply adding a few rules on

how the algorithm reacts on certain block ids. These rules shown in figure 8, resulted in the artificial map produced in figure 3.

```
b = np.random.randint(384)
c = np.random.randint(384)
for x in range(256):
    for y in range(128):
        for z in range(128):
            if (z%32)<16 and (y%32)<16:
                res[x,y,z] = s[x,y+b,z+b]
            elif z>=16 and y<16:
                res[x,y,z] = probability(relam[res[x,y,z-1],0],res[x,y,z-1])
                if x-5 >= 0:####REMOVE BLOCKS IN THE AIR
                    if res[x-5,y,z] == 0:
                        res[x,y,z] = 0
            elif z<16 and y>=16:
                res[x,y,z] = probability(relam[res[x,y-1,z],1],res[x,y-1,z])
                if x-5 >= 0:####REMOVE BLOCKS IN THE AIR
                    if res[x-5,y,z] == 0:
                        res[x,y,z] = 0
            elif (z%32)>=16 and (y%32)>=16:
                res[x,y,z] = s[x,y+c,z+c]
            else:
                res[x,y,z] = probability(relam[res[x,y-1,z],1],res[x,y-1,z])
                if x-5 >= 0:####REMOVE BLOCKS IN THE AIR
                    if res[x-5,y,z] == 0:
                        res[x,y,z] = 0
```

Figure 8: The For loop that creates each initial individual map. Each ID corresponds to the position of the block in the map, so it is used to make position dependent rules.

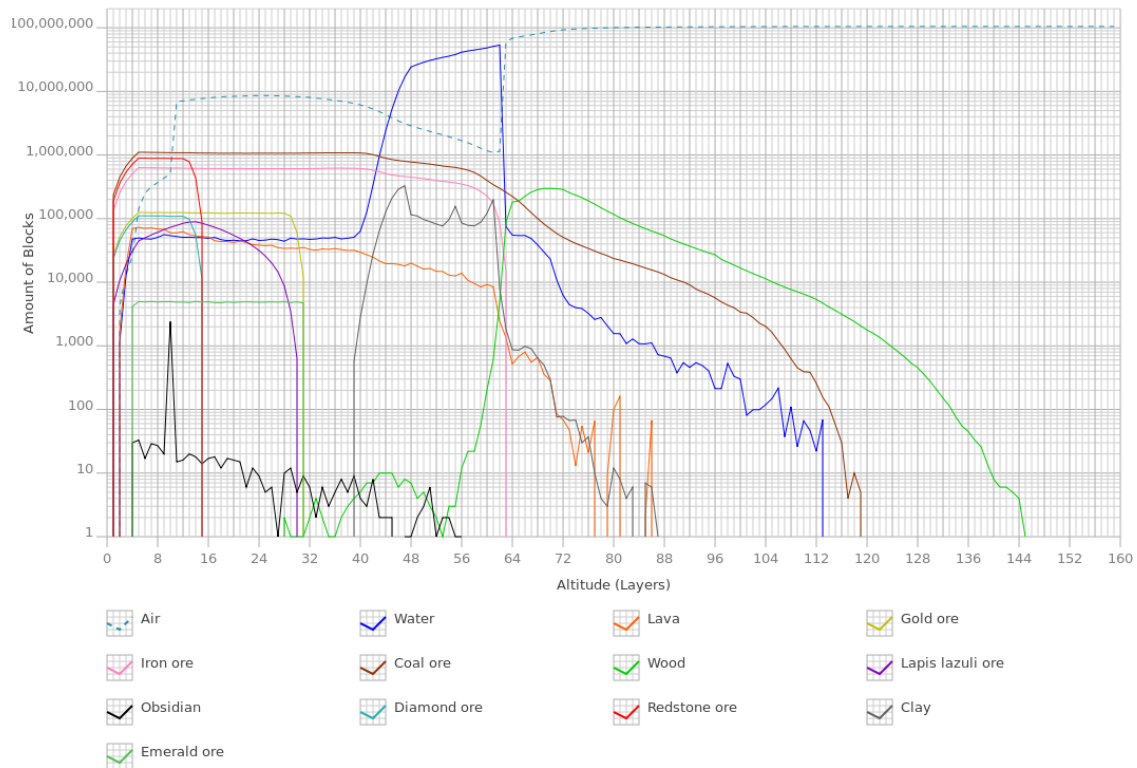


Figure 9: The amount of blocks of each type at each altitude.

6 Limitations and Possible Solutions

Quilting has proven to be a lot more efficient than the genetic algorithm approach, especially because it is a lot easier to optimize. The Numpy library is far more efficient at creating arrays rather than Python, but it only sees benefits when operations applied to the list are vectorized. That is rather than iterating over each element, a Numpy array can change every element at once with the same rule. Since the genetic algorithm approach requires loops in its functionality, this option is not possible.

In the current quilting implementation, I also rely on loops to grow the map by predicting what the current block will be by looking at the previous block. This wastes time when iterating on a sample block that originates from the source because in the loop technique every block must be visited even if it doesn't require searching the hash table. The best way to utilize Numpy effectively here is to decrease the amount of loops required even if it means having to modify more information in the data arrays of the project. By simply selecting a chunk of the source equal in size to the artificial array to be created, we already have a strong base to create a new map. Since this subarray would originate from the source it would take merely seconds. In fact trying the preliminary results of this method can be completed as quickly as 15 microseconds, an enormous improvement in efficiency.

7 Conclusion

Although the procedural generation techniques are far too involved and slow to be used in commercial application, it is useful in building a framework to explore alternatives and optimizations to the techniques detailed in this report. The problem of procedural generation is a balance between making a map that follows the established format rules and time. The idea of incrementally growing a map might not be a proper solution, but the techniques applied such as quilting maybe the first steps into creating more efficient algorithms.

Figure 10: Timing Results:

1. Quilt V2
 - a. overhead: `numpy.unravel_index + numpy.random.choice`
 - i. `x=100` | 0.5940110517198036 seconds
 - ii. `x=1000` | 5.992977313594679 seconds
 - b. overhead: `flatten() + numpy.random.choice()`
 - i. `x=100` | 0.5923405670368993 seconds
 - ii. `x=1000` | 5.89890591258802 seconds

best case: 5.89

$((256*128*128)/1000)*5.89$ seconds = 24704.46 sec = 6.86235 hours per map (64 chunks)

$((256*64*64)/1000)*5.89$ seconds = 6176.11 sec = 1.71559 hours per map (32 chunks)
2. `ngschm.py` - premade 20 individuals, 10 generations w/ fitness
 - a. Elapsed: 828.16 s, Kernel: 3.29 s (0.4%), User: 819.68 s (99.0%)
 - b. Elapsed: 824.69 s, Kernel: 3.17 s (0.4%), User: 817.54 s (99.1%)
 - c. Elapsed: 2249.36 s, Kernel: 741.15 s (32.9%), User: 1501.28 s (66.7%)
 - d. Elapsed: 806.21 s, Kernel: 2.42 s (0.3%), User: 803.03 s (99.6%)
3. `ngschm.py` - full; 20 quilted individuals, 10 generations w/ fitness
 - a. Elapsed: 5852.52 s, Kernel: 2295.82 s (39.2%), User: 3521.21 s (60.2%)
 - b. Elapsed: 3014.72 s, Kernel: 765.59 s (25.4%), User: 2213.51 s (73.4%)
 - c. Elapsed: 4437.92 s, Kernel: 1541.80 s (34.7%), User: 2850.61 s (64.2%)
4. `ngschm.py` - 2 quilted individuals, no generation, and no fitness
 - a. Elapsed: 59.40 s, Kernel: 0.33 s (0.6%), User: 58.55 s (98.6%)
5. `ngschm.py` - 20 quilted
 - a. Elapsed: 606.86 s, Kernel: 0.67 s (0.1%), User: 599.51 s (98.8%)
 - b. Elapsed: 618.29 s, Kernel: 0.67 s (0.1%), User: 610.45 s (98.7%)
6. `ngschm.py` - no fitness, random individuals
 - a. Elapsed: 442.80 s (7 min), Kernel: 1.90 s (0.4%), User: 440.75 s (99.5%)
7. `ngschm.py` fitness, random individuals
 - a. Elapsed: 806.93 s (13 min), Kernel: 3.32 s (0.4%), User: 803.16 s (99.5%)
8. `gschem.py` - no fitness, random individuals (Original)
 - a. Elapsed: 1960.60 s (30 min), Kernel: 13.32 s (0.7%), User: 541.18 s (27.6%)
9. selecting subset of source map
 - a. `pie = np.random.randint(62914560)`
 - b. `source[pie:pie+4194304]`
 - c. 1.519e-05 seconds (15 microseconds)

Acknowledgements: I would like to thank Professor John Reppy for his expert advice and guidance. This project was completed as an independent study for CS 29700. It would not have been possible without Mojang's Minecraft, David Rio Vierra's MCEdit, Thomas Woolford's NBT python library, and Justin Aquadro's NBTEditor.

References

- [1] A. A. Efros, and W. T. Freeman. "Image Quilting for Texture Synthesis and Transfer."
<http://graphics.cs.cmu.edu/people/efros/research/synthesis.html>
- [2] A. A. Efros, and T. K. Leung. "Texture Synthesis by Non-parametric Sampling."
<http://graphics.cs.cmu.edu/people/efros/research/quilting.html>
- [3] D. Vierra et al. *MCEdit-Unified*. <https://github.com/Khroki/MCEdit-Unified>
- [4] T. Woolford et al. *NBT*. <https://github.com/twoolie/NBT>
- [5] J. Aquadro et al. *NBTEditor*. <https://github.com/jaquadro/NBTEditor>
- [6] http://minecraft.gamepedia.com/Seed_%28level_generation%29
- [9] <http://minecraft.gamepedia.com/Altitude>
- [7] <http://minecraft.gamepedia.com/Chunks>
- [8] http://minecraft.gamepedia.com/Region_file_format
- [8] http://minecraft.gamepedia.com/Data_values/Block_IDs
- [8] <http://lethain.com/genetic-algorithms-cool-name-damn-simple/>
- [9] <http://graphics.cs.cmu.edu/people/efros/research/synthesis.html>
- [10] <http://graphics.cs.cmu.edu/people/efros/research/EfrosLeung.html>
- [11] <http://graphics.cs.cmu.edu/people/efros/research/NPS/alg.html>