# Self-Replicating Robots in Minecraft

Gavin Davis, Henry Heyden, Wesley Stone

COMP484 - Introduction to Artificial Intelligence

12/18/2024

## Abstract

In our project, we created self-replicating robots in Minecraft, prioritizing independence and resilience against unexpected circumstances. We used turtles from the mod CC: Tweaked. A self-replicating turtle successfully gathers non-renewable resources from the environment and grows renewable resources from a limited starting supply. It then crafts those resources into a copy of itself, equips that copy with the necessary tools to self-replicate, and releases that copy into the world. We limited the scope of our project to ignore fuel consumption by the turtles, as well as continuity between loaded and unloaded parts of the world. At known critical points, the program has an estimated failure rate of 4.4%. Our program demonstrates the power of hard-coded environment-agnostic behaviors as part of a complex AI system interacting with an unknown, highly variable environment.

## 1. Introduction

For our project, we set out to create a self-replicating robot in Minecraft. To be considered a success, the robot needed to seek out the necessary resources to create another robot, assemble those resources into a new robot, and then instruct the newly created robot in how to perform the same process. These robots would then return to the environment to continue self-replicating. In theory, this would continue until the available resources, either computationally or environmentally, were fully exhausted.

This is an interesting problem for a few reasons. Self-replicating robots have been part of robotics conversations for a long time. In his 1872 satire on Victorian society, Samuel Butler argued that machines would never be able to self-replicate (Butler 1872). In 1966 John von Neumann published a book detailing 2-dimensional cellular automata that could replicate their starting configurations (Schwartz, Von Neumann, and Burks, 1967). Further, in 1986, K. Eric Drexler imagined a world with self-replicating nanobots that could overtake the world in a “gray goo” scenario (Drexler 1990). In 2008, Eliezer Yudkowsky (Yudkowsky 2008) predicted that nanotechnology, combined with self-replicating human uploads, could become the next arms race, in which the advantage of one day might make all the difference.

Because the physical constraints of the real-world currently make it prohibitively difficult to build truly self-replicating machines, most of these discussions have been theoretical. They ask questions about exponential growth, environmental sustainability, and intentional limits on robotic development. By building a self-replicating machine in a complex digital environment, we can interrogate the theoretical groundings of robotic self-replication in a more concrete way.

Furthermore, these machines are relevant to users of the game Minecraft. Many of the players who use these robots use them to autonomously gather resources. If they were to use a program like ours, the questions these philosophers ask would not be thought experiments, but actual steps to protect a player’s progress in-game.

## 2. Technical Background

To implement our vision, we used a Minecraft mod called CC: Tweaked, which is a fork of an older mod called ComputerCraft that has been modernized for newer Minecraft versions (‘CC-Tweaked’ 2024). CC: Tweaked adds computers, monitors, and programmable moving blocks called turtles to Minecraft. Users can interact with these objects using the programming language Lua, which is a functional, dynamically-typed language. That code is stored online somewhere (CC: Tweaked has a built-in API for downloading/uploading code to Pastebin, but also allows for other sources), and then can be downloaded to the block in Minecraft, where the Lua code can be edited and executed by the block.

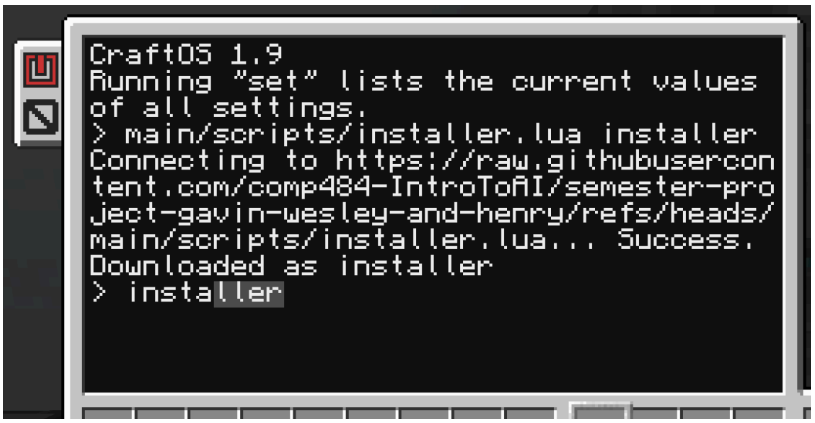



Figure 1. Terminal of a CC: Tweaked Turtle

We used the turtles from CC: Tweaked as the basis for our self-replicating robots. Turtles are recognized by the game as a block, so they are not attacked by hostile NPCs, nor do they break when they run into fire or lava. Turtles can be equipped with up to two items—when a turtle is equipped with a diamond pickaxe, it can break any block other than bedrock and when it is equipped with a crafting table, it can craft any recipe that could normally be crafted in a crafting table by rearranging its inventory to match that recipe. The turtle’s inventory is a 4x4 grid, able to hold up to 16 different types of items, and up to 64 items in any one slot. Turtles can also be given external information; by writing to their ‘settings’ module, developers can save variables to the block’s memory, allowing them to bypass the visibility/continuity of setting variables within Lua.

At any given point in time, turtles have a particular set of percepts. They are able to look at the details of the blocks directly above, below, and in front of them. This means that if they walk past a block that is to their left or right, they will not see it unless they turn to look at it. They can also see a single slot in their inventory, which includes what item is in that slot and how many of that item are there. In addition, turtles can check their immediate surroundings for a ‘periphery’, which is a specific type of block that the turtle can access, like a chest, furnace, or disk drive.


Figure 2. Turtle's vision range and an example of a turtle's inventory (selecting 38 Cobbled Deepslate in slot 9)

The set of basic actions available to turtles is similarly limited. They can move one block forward, backward, up, or down. They can also turn left or right. To manage their inventory, they can move items between slots, and they can drop their selected item into the world or an adjacent inventory. If they are selecting a block, they can place that block above, below, or in front of them. By default, turtles use fuel every time they move, at 1/10 the rate that the fuel is burned in a furnace. A piece of coal, which normally smelts 8 items in a furnace, would allow the turtle to move 80 times.

## 3. Related Work

Using Minecraft as a tool for researching artificial intelligence is not new. Microsoft has a research project called Project Malmo, which is a platform for developing deep-learning AI to control a player-character in Minecraft (Johnson, *et al.* 2016). A group of researchers from Brown University have used Minecraft as a playground for measuring an AI's ability to solve labyrinths in a complex environment (Aluru, *et al.* 2015). Some developers have been using large-language models to create an "open-ended" AI that can explore and gather resources, like our project (Emergent Garden 2024). ComputerCraft has also been used in an academic setting, but the studies were focused on using ComputerCraft as a tool for teaching computer science, not as a tool for implementing artificial intelligence (Wilkinson, Williams, and Armstrong, 2013).

While we did not find any pre-existing academic literature on self-replication using CC: Tweaked turtles, there is a large community of developers for CC: Tweaked. The mod has been downloaded nearly 700,000 times, and is included in many of the most popular technological modpacks, including All The Mods 9, Gregtech New Horizons, and Create: Above and Beyond (*CC: Tweaked - Minecraft Mod* 2021). Many of the turtle programs people write for their local Minecraft worlds are not published in an easily accessible way, but some are.

Notably, this project was inspired by a YouTube video by MerlinLikeTheWizard, which details the way that he created self-replicating turtles in Minecraft (MerlinLikeTheWizard 2022). When we began the project, we saw that he had not published his code on Github, so we believed our project would be a novel source of public code. However, in the final few weeks of development we realized that a link to his code on Pastebin was in the description of the YouTube video.

Merlin's video is a technical demonstration, not a deep-dive into his design philosophy, so many of our practical implementations for resource collection and generation are based on his approach, but none of our higher-level design decisions were derived from his video. As we describe our methods in the next section, we will clarify which portions of our practical strategy align with his and which do not.

As part of our testing and debugging process, we also used a tool that Merlin made to make the Minecraft player automatically follow turtles (MerlinLikeTheWizard 2024).

## 4. Our Approach

One important thing to note about our approach, which is different from Merlin's approach, is that we disabled turtles' fuel usage. We realized early in development that the project's scope of complexity was very large, and that we would need to limit it to finish within our timeframe. Disabling fuel enabled us to focus on the higher-level 'AI' aspects of the problem, instead of on the lower-level optimization aspects. Instead of worrying about whether we were doing any particular step in the most fuel-efficient way, we could focus on just doing the step. Disabling fuel also meant that we would not have to worry about error management. If fuel was enabled, we would constantly need to check whether our fuel had dropped too low, and break out of our current course of action if so. By disabling fuel, we could more comfortably hard-code specific patterns of behavior, assuming that none of the steps would unexpectedly fail.

We also limited the scope of our program to only include turtles that do not leave the render distance of the Minecraft player. In Minecraft, only a certain radius around each player is 'loaded' and anything outside of that range ceases to exist until the player re-enters the vicinity. When this occurs to turtles, they forget where in the operation of their code they were, and start over from the beginning. We chose to ignore this possibility, and only evaluate our program under conditions where it stays within loading-range of the Minecraft player.

Now that we have covered those caveats, let us look at the specific practical problem of turtle self-replication. For a turtle to self-replicate, it needs to craft two main things: another turtle and the tools that the new turtle will need to also self-replicate. These tools include a diamond pickaxe and a crafting table to give to the new turtle so that it can mine and craft. They also include a disk drive and a floppy disk, which the parent turtle can use to pass its code to the new turtle. Both of these steps require the turtle to mine for non-renewable resources underground, grow some renewable resources above-ground, and smelt some of those resources

into more refined versions. The complete crafting trees for both steps are available in Appendix A and Appendix B.

We approached these crafting trees as though they were ‘shopping lists’, where the turtle would need to gather enough of each basic ingredient, and then take a series of processing steps to craft them into our final requirements. We called the resource-gathering phase the Mining stage, and the processing phase the Crafting stage. The complete list of required basic ingredients is shown below in Figure 3. The idea for separating the process into basic ingredients and processing steps came from a paper about self-replication in real-life robots (Lee, Moses, Chirikjian 2008). They broke the process of self-replication into three parts: gathering ‘base’ resources, applying processing steps to those resources, and using any pre-existing environmental features to complete the process. The level of external assistance in any of these three areas can determine the ‘degree’ to which a system is self-replicating.

Figure 3. Full set of required ingredients before final crafting step

In the mining state, the turtle first mines for each of the non-renewable resources it can find underground. The strategy for this is simple - go to the height level that the resource appears at most frequently, then mine in a straight line until the turtle encounters it. This requires the turtle to know its height level, which it does not know inherently. To address this, we have the turtle ‘calibrate’ by traveling down until it hits bedrock, which generates at the bottom of every Minecraft world, and then we keep track of any changes in height after that. This is basically the same process that Merlin uses in his approach. Because the turtle needs every resource before it can move to the next state, and none of the resources let the turtle get the other ones faster, there is no required priority for resource-gathering—instead, we arbitrarily set our priority to go from bottom-up, assuming that we are generally starting to look for resources being at bedrock.

Once the turtle has gathered all of the necessary non-renewable resources, it enters the planting sub-state of crafting. In this phase, it plants all of the renewable resources it needs.

These include sugarcane and birch saplings, which we give to every turtle upon creation. This process involves finding water, placing down sugar cane, and then waiting for that sugarcane to grow until it has enough to satisfy its sugar cane requirement. Once it has enough sugarcane, it travels to the top of the Minecraft world and places down a sapling. It waits for the sapling to grow, gathers the tree, then repeats until it has enough wood and saplings to satisfy its recipe. In Merlin's implementation, he does not carry around a sapling and grow it, but instead searches the Minecraft world for pre-existing trees and mines them instead (MerlinLikeTheWizard 2022).

After the turtle finishes growing its renewable resources, it enters the processing sub-state of crafting. This sub-state starts by placing down a chest that the turtle has been carrying around to store its inventory while it does the processing. It then crafts enough wooden planks to use as fuel for smelting, then a furnace, and then it smelts each of the resources that need to be smelted. Then it progresses through its crafting trees in a hard-coded pattern until it ends up with a new miney-crafty turtle, disk drive, and floppy disk. At this point its inventory contains one sugar cane, two birch saplings, and one chest to give to the new turtle.

After the processing sub-state, the turtle enters the reproduction sub-state. This sub-state is a hard-coded series of actions that will be performed the exact same for every turtle under every circumstance. In this sub-state, it places down a disk drive, puts a floppy disk into it, and then writes a copy of its code onto the floppy disk. Then it places the child turtle next to the disk drive, and gives it the sugarcane, saplings, and chest. It then turns the child turtle on, triggering the child to run a script that downloads all of the code from the floppy disk. This process aligns with Merlin's implementation as well. The child turtle then leaves to begin its own self-replication, starting by calibrating its height level. After the child leaves, the parent gathers its remaining inventory from the chest it placed down earlier, drops any items that are not 'base' resources, and returns to the Mining state.

On a theoretical level, we conceptualized the process of moving between these Mining and Crafting states as a hierarchical finite-state machine, where the robot had the two main states of behavior, as well as a few internal states within them. The code also has a few hooks that move the turtle between states depending on a series of environmental factors. Notably, it moves to the crafting state from the mining state once it has 'completed' its shopping list. It also moves to the processing sub-state from the planting sub-state once it has gathered enough plant resources. The complete finite-state machine can be seen in Figure 4.

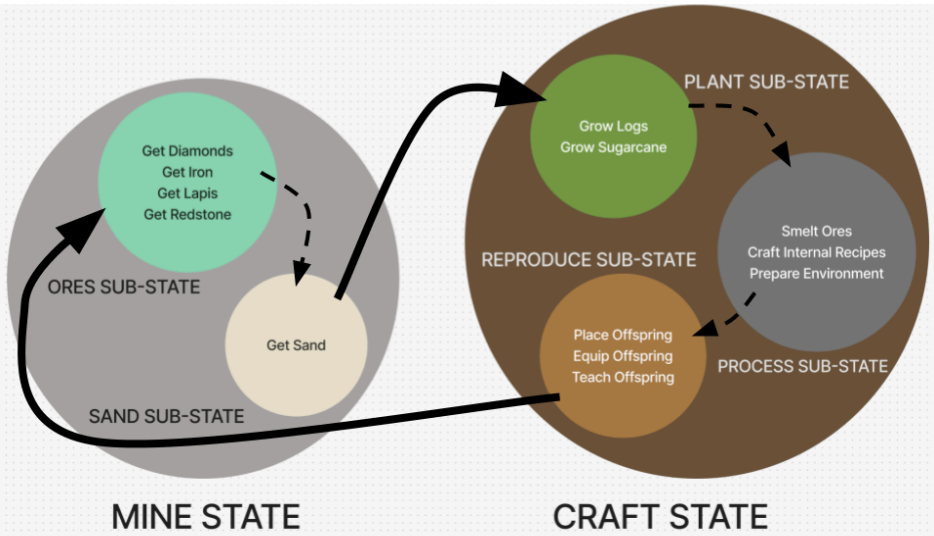



Figure 4. The conceptual finite-state machine

Within each state, the turtle operates more like a reflex agent. For instance, when the turtle is mining for resources, it just repeats the same reactive steps every time-step: it moves forward, checks the blocks above/below/in front of it for the ore, and runs a function to mine the vein of ore if the desired ore is there. Within these steps, the turtle has no memory of its previous actions, nor any higher-level concept of a 'shopping list'. It only retrieves its percepts, reacts immediately, then continues, sending a message to the higher-level finite-state machine when it is complete. This also applies within the planting sub-state, where the turtle just places the crop, checks whether it has grown yet, and runs a function to gather the crop if it has grown.

## 5. Analysis

Our code works well, given the constraints we placed on it. Because we did not have to worry about fuel or chunk-loading, the list of fail conditions (where a turtle is not able to successfully self-replicate) are fairly limited. They occur in two expected places: bedrock calibration and sapling replacement rates. They also occur in a variety of unexpected places as a result of bugs or programming oversights. Let us start by looking at the two expected fail conditions.

The first is bedrock calibration. This step is critical, because the turtle needs to have an accurate picture of its height level to gather non-renewable resources correctly. The problem is that there are five layers of bedrock, from -60 to -64, starting at a ⅕ concentration of bedrock and increasing by that amount with each layer down (-64 is entirely bedrock), so the first time the turtle encounters bedrock could be at any of these depths.

In theory, we could address this by having the turtle automatically move up a level any time it accidentally encountered bedrock, but this would require a significant restructuring of our program to allow in-operation error management. Many of our functions rely on the assumption

that the turtle and the world's state are exactly as they were before the function, minus whatever changes the function is actively intended to make. Instead of doing this, we implemented a manual process where the turtle checks its surroundings for more bedrock when it first encounters bedrock, in the hopes that we can minimize our chances of misidentifying our height level.

Initially, we had the turtle move into the four adjacent blocks, which would allow it to see those as well as their eight neighbors. But this introduced a possibility of failure—only one out of every five blocks on the top level is bedrock, so there was a 7% chance that the turtle would incorrectly determine its height-level, even when checking 12 blocks. Therefore, we had the turtle go one block further in each direction, doubling its total blocks checked, and reducing the chance of failure to $(\frac{4}{5})^{24}$, or 0.47%.

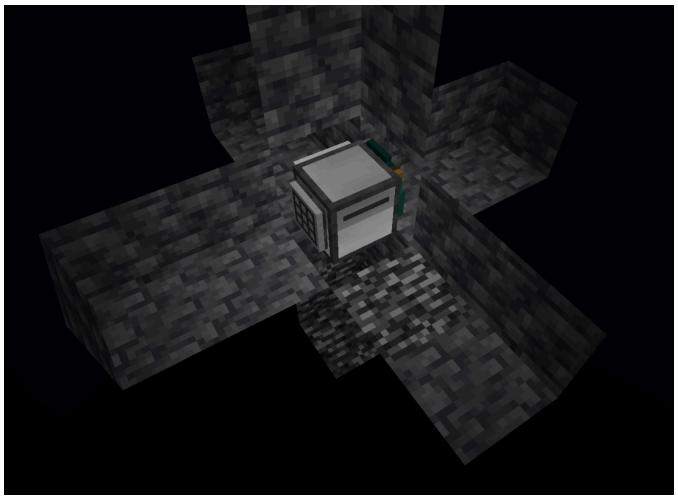


Figure 5. Turtle running bedrock calibration

The second expected failure point occurs when we grow our trees: there is a chance that they do not produce enough saplings to replenish our supply. Merlin bypassed this problem by having the turtle find trees in the world. Technically, this is a better solution for our approach, where we have no limitations on movement. But we wanted to distinguish our approach from Merlin's. We also found that locating trees in the world is much trickier than it is for ores: tree's locations are greatly dependent on biome, there is no consistent y-value at which they grow, and the density of their trunks along the ground is very low, so it would substantially increase our program's run-time.

To address problems surrounding sapling replacement rate, we created a probability distribution to determine the chance of failure. Each leaf in a tree has a 5% chance to drop a sapling. This means that we could get many saplings from a single tree – or, conversely, none at all. To find out, we defined a worst-case probability distribution for the number of saplings the turtle can harvest from a birch tree:

$$\sum_{n=0}^{50} (0.95^{(50-n)} \cdot \ 0.05^{n} \cdot nCr(50, n))$$

This is worst-case because 50 leaves is the minimum that can grow on a birch tree (the maximum is 60). Keeping in mind that these numbers are slight overestimates, we found that each tree has approximately a 7.7% chance of spawning zero saplings, which would cause our program to fail. Also, there is a 20.2% chance of only getting one sapling from the tree, even though the turtle should come away with two saplings, one for itself and one for its child.

The fix was relatively simple. By simply giving the original turtle an extra sapling to start with, we reduced the chance of total failure in the first run to about 0.6%. Additionally, there is only a 3.2% chance of it coming away with just 1 sapling. Conversely, there is a 70% chance that the turtle gets at least 6 saplings, so on average the number of saplings is well above replacement and increases with time.

Discounting these edge cases, and conservatively counting every bare replacement scenario as a failure, we arrive at a 3.8% chance of failing here. Combined with our bedrock procedure, we have a total estimated 4.4% chance of failure in the first run. While this is an estimate, it is largely based on conservative assumptions, and so far, theory has aligned with practice – in the relatively few full runs we have done, neither bedrock nor saplings have failed. Therefore, we are satisfied with how our code approaches our expected failure conditions.

This brings us to the unexpected failure conditions, which would arise from bugs in our program, unexpected world generation, or programming oversights. The most obvious unexpected failure condition comes from dirt and cobblestone. Our program requires one dirt block and at least 22 cobblestone to properly run, but it doesn't have any procedure to explicitly mine for them, assuming that it will incidentally run into enough of them to have extras when it begins crafting. There are plenty of circumstances where that might not happen, and if we had more time, we would add them to our list of required resources in the Mining state and implement solutions to gather more.

Another unexpected failure condition can come from interaction between turtles. When there is only one turtle in the world, we can assume that the environment's state will remain the same between action steps. However, when multiple turtles start to interact with the same space, those assumptions don't work anymore. If a child turtle were to begin its hard-coded crafting process in the exact same space as the parent turtle, it may accidentally break blocks that the parent assumes are there, and because our code does not do any error-checking of its assumptions about the world, it would break instead of recalibrating.

There are also plenty of places where our code could operate better, even though it does not cause total failure. For instance, if turtles run into a resource they need, but they are mining for a different resource, they will not opportunistically gather the unexpected resource to prevent them from needing to look for it later. This does not make sense, and can significantly increase the time it takes for the program to run. They also have to wait for daylight to grow trees,

because the trees have a minimum light level to grow. In theory, the turtles could carry around a torch to always have enough light to grow the turtles. This is important because night-time lasts ten minutes, which are ten minutes where our turtle is guaranteed to be making no progress.

The time it takes for our program to run is extraordinarily relevant, because it directly correlates with how easy it is to test. We found that each run of our program took about an hour, even when we artificially sped up plant growth and maintained daylight. However, we also know that bugs would occur most commonly when the turtle ran into unexpected world generation, which could only be manually tested by running the program under different circumstances.

## 6. Future Work

As has been mentioned in §3 and 5, we disabled the fuel requirement of the turtles to reduce the scope of this project. A natural future goal of the project is thus to reintroduce this restriction to the turtles, and make the necessary modifications that follow. This decision would necessitate that the turtle could pause what it is doing at any point and transition to refueling.

Another future implementation goal which deals with the resiliency of our turtle is dealing with the situation in which a turtle leaves the player's vicinity and becomes unloaded. At present, the behavior of the turtle when reloaded is to completely restart the process from the beginning. The optimal behavior would be for a reloaded turtle to be able to figure out what step of the process it was performing before being suddenly stopped, and seamlessly continue that process.

Reimplementing these two features opens room for much more work. First, it would let us engage with our theoretical idea of a finite-state machine better. One of the main benefits of finite-state machines is that they can transition between behavior in a nonlinear, seamless way. However, under our current implementation, the turtle moves through the states in a fairly linear, unchanging way. Reimplementing fuel would justify the finite-state machine description by giving new hooks to switch between states in a more non-linear way, and being able to identify the current state and re-enter it after being reloaded would do the same thing.

It would also improve the accessibility of our program. Under current circumstances, our code would be useful for a very small subset of players, because they would need to manually change the configuration settings for their world to disable fuel, and they would need to constantly stay near the turtles to keep them operational. Once we address those two issues, our program would become the public source of assistance that we originally hoped it would be.

Further, once we reintroduce fuel requirements, we open the door to lots of optimization within our program. Right now, there are no consequences for inefficient behavior, other than longer program run-times and more difficult debugging. Once fuel is re-enabled, any inefficiency directly correlates to a higher chance of failure. There are many places where this optimization could occur.

For instance, when the turtle finishes replicating, it leaves behind many of the tools it created as part of the process. If it could hold onto some of them, it could potentially save a lot of

time and movement. Notably, it should only need to gather sand once per every sixteen replications, because the six sand it finds produces enough resources to craft sixteen turtles. Also, opportunistically mining unexpected ores becomes much more important, because the turtle is now minimizing fail conditions, not just saving time.

Once these goals are successfully implemented, experimentation can be done on the behavior that emerges when a multitude of turtles interact in the Minecraft world. Once there are two, three, or more turtles in the world, is there any emergent behavior? How do they respond to the decreased amount of resources available where they begin, and what happens if a turtle's reflex is to mine another turtle? These are questions that can be answered in time, and any suboptimal behavior can subsequently be finetuned in the code base.

## 7. Conclusion

In this project, we successfully created self-replicating robots in Minecraft using turtles from CC: Tweaked. These robots rely on a mix of high-level finite-state-machine behavior and low-level reflex agent behavior. We tactically employ hard-coded behavior that lets the robots perform complex behavior in highly varied environments. The robots have a greater than 95% estimated success rate at self-replicating on their first iteration, improving with each subsequent iteration as they build a buffer of extra resources.

However, this success occurs with a heavily limited scope of potential failure conditions, because we disabled fuel-consumption and chunk-loading. This severely limits the accessibility of our program, as well as its potential use-cases. It also enables the program to ignore many forms of optimization, because they do not directly impact the program's success rate.

There are many potential avenues for future work on this project. The first priorities would be to re-implement and manage our two major limitations: fuel-consumption and chunk-loading. Once that occurs, it would open the door for many optimizations to minimize the plethora of new failure conditions that would be introduced. These avenues would also provide more ways to use this project as a theoretical testing ground for self-replicating robots elsewhere in the world, which could include questions about internal limits on self-replicating AI, energy use, and exponential growth.

## 8. References


Aluru, K.C. *et al.* (2015) 'Minecraft as an experimental world for ai in robotics', in *2015 AAAI Fall Symposia, Arlington, Virginia, USA, November 12-14, 2015*. AAAI Press, pp. 5–12. Available at: http://www.aaai.org/ocs/index.php/FSS/FSS15/paper/view/11725 (Accessed: 17 December 2024).

Butler, S. (1872) *Erewhon*. Available at: https://www.marxists.org/reference/archive/butler-samuel/1872/erewhon/ch24.htm (Accessed: 17 December 2024).

*CC: Tweaked - Minecraft Mod* (2021) *Modrinth*. Available at: https://modrinth.com/mod/cc-tweaked (Accessed: 17 December 2024).

'CC-Tweaked' (2024). CC: Tweaked. Available at: https://github.com/cc-tweaked/CC-Tweaked (Accessed: 17 December 2024).

Drexler, K.E. (1990) *Engines of creation: the coming era of nanotechnology*. 6. [print.]. New York: Doubleday (Anchor books).

Emergent Garden (2024) *AI Plays Minecraft Forever (and dies)*. YouTube. Available at: https://www.youtube.com/watch?v=IeXadWbvDiE (Accessed: 17 December 2024).

Hoang, H., Lee-Urban, S. and Muñoz-Avila, H. (2021) 'Hierarchical plan representations for encoding strategic game ai', *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 1(1), pp. 63–68. Available at: https://doi.org/10.1609/aiide.v1i1.18717.

Johnson, M. *et al.* (2016) 'The malmo platform for artificial intelligence experimentation', in. Available at: https://www.microsoft.com/en-us/research/publication/malmo-platform-artificial-intelligence-experimentation (Accessed: 17 December 2024).

Lee K, Moses M, Chirikjian GS. Robotic Self-replication in Structured Environments: Physical Demonstrations and Complexity Measures. The International Journal of Robotics Research. 2008;27(3-4):387-401. doi:10.1177/0278364907084982. Available at: https://www.researchgate.net/publication/220122898_Robotic_Self-replication_in_Struct

ured_Environments_Physical_Demonstrations_and_Complexity_Measures (Accessed 17 December 2024)

MerlinLikeTheWizard (2022) *Self Replicating Turtles in Minecraft ComputerCraft.* YouTube. Available at: https://www.youtube.com/watch?v=MXYZufNQtdQ (Accessed: 17 December 2024).

MerlinLikeTheWizard (2024) 'tfollow'. Available at: https://github.com/merlinlikethewizard/tfollow (Accessed: 17 December 2024).

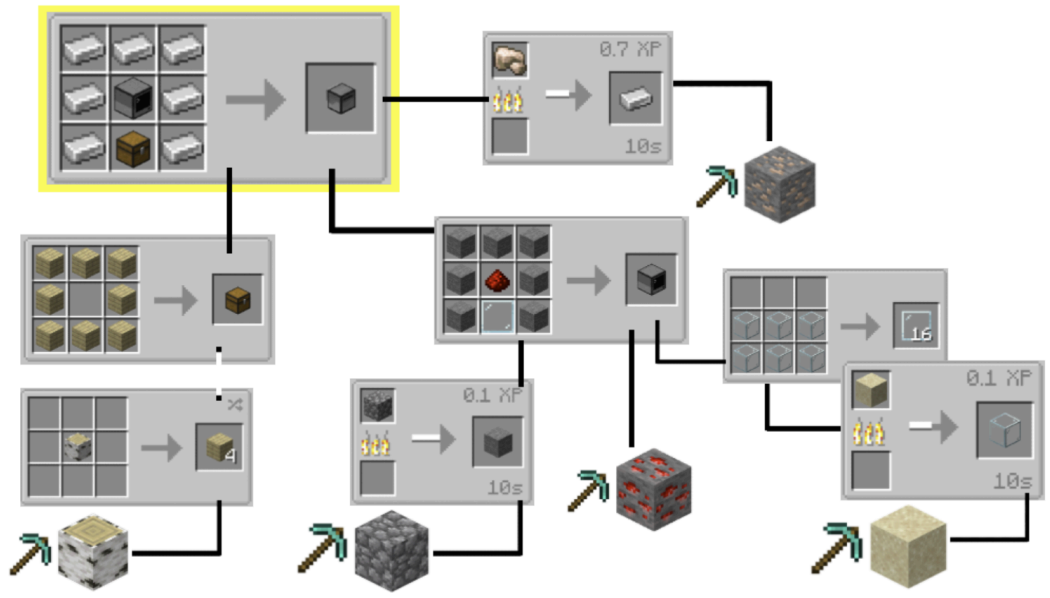
Schwartz, J.T., Von Neumann, J. and Burks, A.W. (1967) 'Theory of self-reproducing automata', *Mathematics of Computation*, 21(100), p. 745. Available at: https://doi.org/10.2307/2005041.

Wilkinson, B., Williams, N. and Armstrong, P. (2013) 'Improving student understanding, application and synthesis of computer programming concepts with minecraft: ectc2013', in.

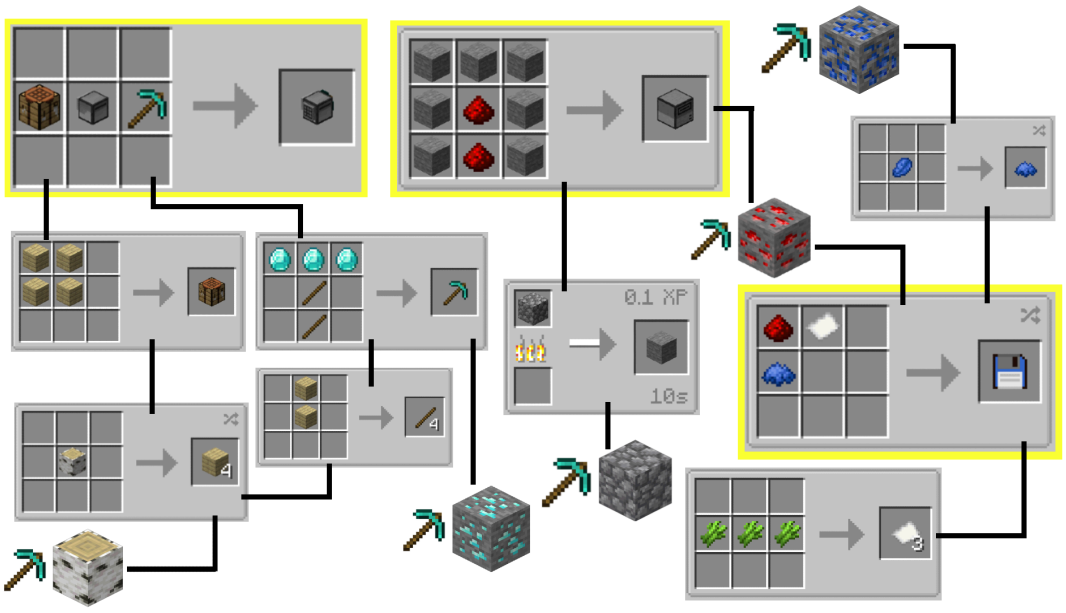
Yudkowsky, E. (2008) 'Total nano domination'. Available at: https://www.lesswrong.com/posts/5hX44Kuz5No6E6RS9/total-nano-domination (Accessed: 17 December 2024).

Any pictures of Minecraft blocks or items used in this paper are sourced from either a screenshot we manually took, or from the Minecraft wiki

## 9. Appendix




Appendix A. Crafting tree for a turtle




Appendix B. Crafting tree to create disk drive, floppy disk, and equip turtle