Universiteit Leiden
The Netherlands

Opleiding Informatica

BlockStep, an A* algorithm toward Minecraft optimal world traversal for speedrunning

Isaac Braam

Supervisors:
Dr. R. van Vliet & Dr. J.N. van Rijn

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

30/08/2022

Abstract

Minecraft is a popular game in which many players attempt speedruns. In this thesis, we discuss how we could use a reinforcement learning agent to break the world record. This agent will need to build on pre-defined optimal strategies to reduce the complexity of analysing the game world. This is possible, because Minecraft is partly deterministic. Parts of the strategies rely on an optimal path traversal algorithm. In this thesis, we describe the developement of a near-optimal version of the A* algorithm for path traversal in the Minecraft block world. The previous best algorithm for this task, named Baritone, also uses A*. An obvious disadvantage of Baritone, however, is the restricted movement. A player in Minecraft can turn in an infinite number of degrees. Baritone's creators limited the turns to 45° intervals to make the algorithm less complex. Movement restrictions lead to less optimal paths. We would expect faster paths to be found by expanding the movement space. In 85% of the cases tested, we find that our hypothesis is correct and that our new BlockStep algorithm finds a faster path than Baritone.

Contents

Chapter 1

Introduction

1.1 Computer advancements in games

Since computers have been invented, people have wondered how computer intelligence compares to human intellect. We have several methods for measuring intelligence, such as the Turing test [M.F00] and IQ tests [DHO12]. Games are also a good way to measure the efficiency of computers. For most games we often aim to make a program that is capable of beating the game's experts. Over the years, computer programs have claimed numerous victories over their human counterpart in complex games such as Chess [CJhH02] and Go [SHM+16]. The latest computer programs even dominate some digital games with nearly infinite game states, such as League of Legends and Starcraft [Vin19]. Minecraft is a game that has yet to be won by a computer.

1.2 Minecraft, and how to beat it

Minecraft is a block-based game in which you can alter the world by placing and destroying blocks to accomplish goals. The game comes with no predefined end goal. The community, however, does agree on a single goal to finish the game: killing the 'Ender dragon.' A run is the amount of time it takes to kill the Ender dragon after starting a new world. The current world record run clocks in at just under ten minutes [Sha22]. AltoClef [Mat22], an autonomous computer program has completed the game in less than three hours, which is comparable to the time required by most human beginners. This program, however, had access to game data that it is regularly not supposed to have access to, resulting in the program having the ability to see through obstacles, which is an ability that regular players do not have.

While researching programs with the same objective of completing Minecraft as fast as possible, we found that these programs consisted of different separate algorithms, layered on top of each other. The pathfinding algorithm served as the foundational algorithm for the construction of these programs. We found that there was no pathfinding algorithm capable of traversing land at a world record pace. To defeat humans in the game of Minecraft, a more optimized algorithm was required.

There is one pathfinding algorithm that is commonly used in Minecraft. The name of this algorithm is Baritone [Cab22]. The algorithm uses A* to calculate the shortest path. The algorithm is well optimized as 54 people have worked on this over the past four years. The aspect of the algorithm that could be

improved is the number of possible walking angles. The algorithm did not utilize more than eight angles as the creators had to work around a problem found when executing a path. This issue stems from the way that Minecraft is programmed. The Minecraft player input is updated every 50 ms. This means that when the algorithm instructs the Minecraft character to jump, the jump will take between 0 and 50 ms to execute. To be sure that the player will land safely on the target block, the instructions for when to jump will have to be an area instead of a simple point. For each angle, these areas would be distinct and often costly to calculate. The algorithm's designers circumvented this issue by using only the eight safest angles to traverse the path.

Our hypothesis is that an algorithm should be capable of performing more complex jumps. These complex jumps are possible if we calculate the safe lift-off area for each of the different angles. These intricate jumps will result in more than the formerly designated eight angles, thereby shortening the path.

1.3 Layout of the thesis

In Chapter 2 we examine the current state of Minecraft speedrunning AI agents.
In Chapter 3 we examine the current speedrunning strategies employed by the world-record holder, from this we deduce a method of creating an AI agent capable of achieving a world record run. With this deduction there are a lot of probability calculations and simulations, as the ability to accurately weigh chance against speed is crucial if you wish to beat the world record.
In Chapter 4 of the thesis we construct the hypothesized path traversal algorithm.
In Chapter 5 we compare our algorithm to the best algorithm currently available.

Chapter 2

Theoretical study on a speedrunning AI agent for Minecraft

2.1 State space

The act of completing a game as fast as possible is known as speedrunning. The Minecraft world is generated based on a random seed, which is an integer value that, when inputted into the algorithm for world generation, generates a unique world. When speedrunning Minecraft, it is required to generate a world's terrain using a random seed. The terrain is generated using Perlin noise [Wik]. Due to the chaotic nature of Perlin noise, it is impossible to infer the appearance of the world by analyzing its surroundings. There are 18,446,744,073,709,551,616 (2^{64}) possible seeds, each of which results in a world that is completely unique. The probability of encountering the same world twice is astronomically small. Therefore, we will have to begin the run without any prior information. If the probability of discovering the same world twice was sufficiently high, the solution to speedrunning Minecraft would be to fully explore a specific world, design an optimal strategy for it, and then to rerun the terrain generator until that same world was discovered again.

The location of the Ender dragon will be based on the world seed. This results in the exact location of the Ender dragon being unknown, which is significant given that the world is essentially infinite. However, the location of the Ender dragon does have specific limitations. One of these limitations is the fact that the location of the Ender dragon can not be within a thousand-block radius of the player's starting point and can not be more than two thousand blocks away. Information like this is crucial for an AI agent because it will narrow the search space from infinite to finite.

The state of the player does not only consist of world and player-positioning data, but also of the items a player possess. A player can have a maximum of 36 distinct items with him at a given time. All the different items a player can possess are shown in Figure 2.1.

2.2 Action space

Not only does Minecraft have a huge state space, it also has an enormous action space. The action space is broad as there are so many items in the game, each with potentially advantageous uses. For the AI

agent to find out autonomously which items are good to have at which given moment seems impossible for the current state of reinforcement learning (RL). To aid the AI agent in reducing complexity, we could restrict the action space to only the items used in the current world record, as these could be deemed the most essential. This list is dissected in Chapter 3 of this thesis.

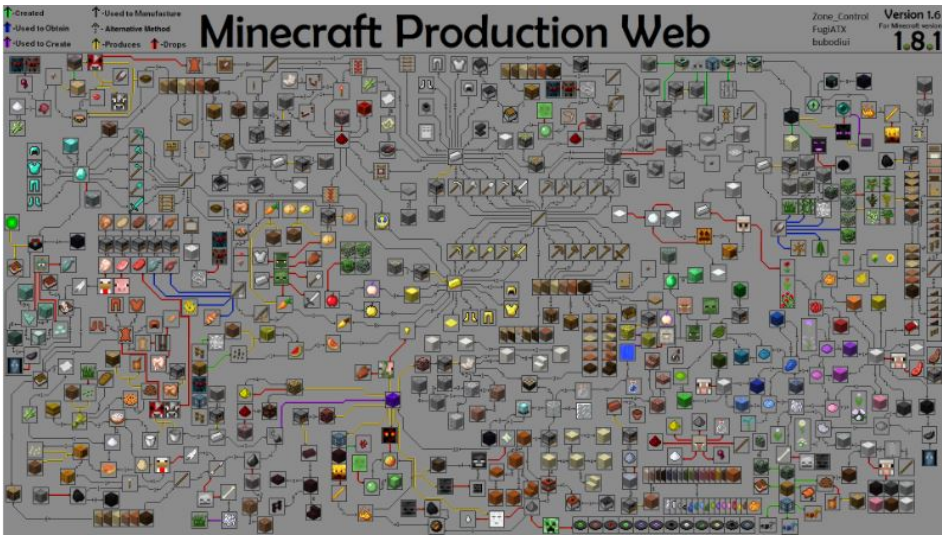


Figure 2.1: This image displays the large quantity of Minecraft items. Items can be turned into other items during the run. The item transformation is often irreversible, this means that a lot of planning and knowledge of the game is needed for an AI agent to be able to finish the game efficiently. The image is from a 2014 version of Minecraft. As the images became too intricate in newer versions of Minecraft, the creators ceased creating them.
Source: www.minecraftworld.wordpress.com.

2.3 Reward

Reinforcement learning is propelled by reward. When the AI agent takes a beneficial step toward the goal, it is rewarded. If the dragon is killed, the speedrun is successful. Because this event is extremely unlikely, an AI agent will almost certainly never learn how to achieve this through simple exploration. Getting to the dragon would require a series of very precise actions.

To assist the AI agent in learning, we will need to set specific subgoals to finish the game. This is exactly what the bot AltoClef does. The bot has been given a detailed sequential plan of subgoals that, if followed in that order, could result in a victory. However, this plan includes steps that are not necessary to finish the game, resulting in the bot never being able to beat the world record. When assigning goals, we would need to create a minimum requirements list. We could look at the current world record and see what items the world record holder gathered to finish the game. From there, we can deduce a minimum required list of items for our AI agent as well as a minimum list of subgoals. These requirements are small tasks with clear goals that our AI agent could train on.

2.4 Related work

In 2019, MineRL hosted a competition in which participants created their own versions of a Minecraft AI agent [Skr22]. The greatest accomplishment of the winning AI agent was the acquisition of iron. Retrieving iron is an important step towards completing the game, but as we will see in Chapter 3, it is only one of many. The winning AI agent in this competition used a variation of the Q-learning algorithm.

Parallel to the work of this thesis a new AI agent for Minecraft was released to the public, which is based on a custom-fit neural network architecture. This AI agent is called MineDojo, and is described in [FWJ+22]. This paper represents a significant advancement in Minecraft AI agents. An AI agent capable of speedrunning Minecraft is now within reach.

The AI agent MineDojo can perform a large variety of tasks. It learned how to perform these tasks by looking at Youtube video's, Reddit posts, and Minecraft wiki pages. This AI agent can complete all of the individual goals in the goal hierarchy that leads to the defeat of the Ender dragon. The minimal subgoal structure that is dissected in Chapter 3 of this paper can easily be integrated into the knowledge base of the agent. With this minimal subgoal structure, the AI agent could autonomously finish the game in a fast pace.

MineDojo was not created with speedrunning in mind, this can be seen in how carefully the agent approaches some tasks. Lava is dangerous in Minecraft, but can easily be avoided, as the lava flow is deterministic. The AI agent still tends to walk very slowly around lava, while it could safely traverse the path in a faster manner.

The AI agent is not rewarded for speed, as its success is solely judged by the accuracy with which it completes a job. This results in the AI agent not having a motive to move fast. When speedrunning Minecraft, we do value speed. To increase the speed with which the AI agent completes its tasks, we can recommend a technique also used by the AlphaGo AI [SHM+16], which is self play. We can let the AI agent learn to be fast by making it complete tasks repeatedly, with small variations in the AI on every run. The AI agent can now be rewarded more as he finishes quicker, in this fashion the AI agent will favor to execute actions in a faster manner where this is possible.

Alternatively we could include the pathfinding algorithm constructed for this thesis into the knowledge base of the AI. In the example mentioned above, this would result in the AI agent knowing that it could get to the lava safely in a faster manner.

Chapter 3

Inferring a subgoal structure for a Minecraft speedrunning AI agent

In this chapter, we derive the minimum requirements for finishing the game, which may serve as a guide for a speedrunning AI agent. The list consists of items observed in the most effective speedrunning strategy found to date. For some items in our minimal requirements list, the number found will depend on a few probability calculations. These calculations will depend on the return distributions of these specific items. For each version of Minecraft these distributions differ. In our deduction we will use the return distributions found in version 1.16.1.

3.1 Items used in the world record run

The video of the current world record is available to the general public [Cub22]. By collecting data from the video we were able to put together the following list of items that were used in the world record run.

- 51 building blocks
- 17 iron
- 7 logs of wood
- 1 flint
- 10 lava source blocks
- 90 gold ingots
- 4 cooked pork chops
- 23 obsidian
- 36 string
- 19 Ender pearls
- 6 blaze rods

- 3 glow stone

When the AI agent has collected all these items, we will be certain that, in theory, we could finish the game in a random world. This does not necessarily mean we will win the game. When only gathering the items on this list, we might not have enough wood to make all the required tools. These tools, for instance, may be necessary if we need to destroy obstacles that could block the entrance to the Ender dragon.

A significant proportion of the speedrun depends on chance, as we will see in detail as we infer the minimum requirements list from the list of items used in the world record run.

At some point we have to build a so-called Nether portal in order to finish the game. You must go through this portal and then accomplish more tasks on the other side. To make the reasoning for the minimum requirements list more clear we can split up the run in pre Nether portal and post Nether portal. Building this Nether portal requires specific items. This method of list division will produce a temporally structured reduction of the current list.

3.2 Pre Nether portal

A Nether portal is a portal that allows the player to travel between the Nether dimension and the normal (Over)world. While speedrunning, a Nether portal's main purpose is to grant you access to the Nether dimension. In this dimension we can find the ingredients needed to make eyes of Ender, which are required to open the gate to the dragon.

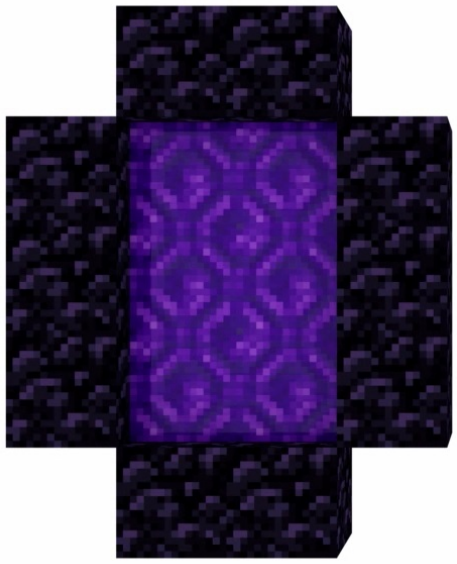


Figure 3.1: The constellation of a Nether portal that requires the smallest amount of blocks.
Source: minecraft.fandom.com.

To build the Nether portal we need the following items:

- 4 iron
- 1 flint
- 1 wood log
- 10 of either lava source or obsidian blocks

We came to this list as we need three iron for a bucket. This bucket is to move the lava source blocks into the pattern shown in Figure 3.1. We could also replace the lava source blocks with obsidian, but the likelihood of finding that much obsidian in the time of a world record speedrun is too small. This is one of the accumulating risks we cannot afford to take, as we will take more significant time-saving risks elsewhere. In order to make the bucket we need the wood to make a crafting table. The remaining one iron and flint are used to create the flint and steel tool, which is used to light the portal in order to activate it.

3.3 Post Nether portal

Once we have access to the Nether, we have three parallelizable tasks:

1. Require enough eyes of Ender to enter the end portal.
2. Gather enough items to get to the Ender dragon in a fast manner.
3. Obtain enough items to kill the Ender dragon.

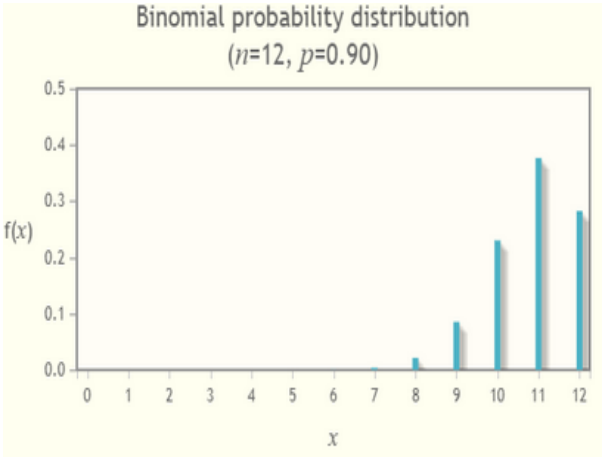
3.3.1 Eyes of Ender

The minimal number of eyes of Ender theoretically necessary is zero, albeit obtaining zero eyes of Ender is highly unlikely. The gate to the dragon consists of 12 slots, each slot requires an eye of Ender to activate. We will call these slots "eye of Ender holders." All twelve eyes of Ender holders must be filled in order to activate the portal. There is a 10% chance that each holder already contains an eye of Ender. Since the probabilities that each holder contains an eye of Ender are independent of each other, we can apply the binomial distribution to calculate the chances of needing exactly x number of eyes of Ender. Let X be the stochastic variable representing the number of empty eyes of Ender holders, i.e., the number of eyes of Ender that we need to gather. Then the probability that X equals a given number x is

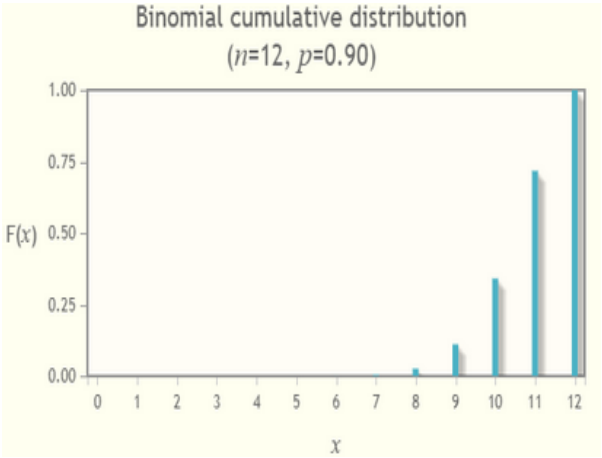
$$P(X = x) = \binom{n}{x} p^x q^{n-x}$$

here, p is the probability that the holder will not already contain an eye of Ender, $q = 1 - p$, this is the inverse probability of p, n is be the number of eyes of Ender holders. When we accumulate these results ($P[X \leq x]$) we get the chance of being able to enter the portal, if we have gathered x eyes of Ender. The precise distribution, and accumulation of the distribution can be found in Table 3.1. Graphs depicting these probabilities are shown in Figure 3.2. When we have zero eyes of Ender, our chances of having enough eyes to go through the portal are 0.1^{12}. This is a chance of 1 in $1 \cdot 10^{12}$ of encountering a world

like this. Worlds like this do exist, as there are $1.8 \cdot 10^{19}$ possible seeds. We would expect each version of Minecraft to have on average $1.8 \cdot 10^{7}$ seeds which require zero eyes of Ender. This may seem to be a lot, but the chance of finding one of these on a random seeds is still $1 \cdot 10^{-12}$, and even when we find one, it will not guarantee a world record on perfect play.



(a) Chances of needing the exact amount of eyes of Ender to get through the portal.



(b) Chances of getting through the portal while having the given amount of eyes of Ender.

Figure 3.2: These graphs were made using an online Binomial distribution calculator.
Source: www.di-mgt.com.au/binomial-calculator.html.

To have an acceptable chance of 2.56% to have enough eyes of Ender we should gather a minimum of eight Ender pearls. The explanation of why we chose the arbitrary cut-off point of 2.5% can be found in Sections 3.6 and 3.8.
The eyes of Ender are crafted out of two parts: blaze rods and Ender pearls in a ratio of 1 to 2 respectively. For 8 eyes of Ender we need 4 blaze rods and 8 Ender pearls.

3.3.2 Blaze rods

The gathering of blaze rods is quite straightforward. You kill creatures that are called blazes, which are likely to be found near each other. We may want to gather more than 4 blaze rods, as the time investment of this is frequently very low. For instance, 5 blaze rods allow for 10 eyes of Ender to be crafted. Hence, as we can see from Table 3.1, having 5 blaze rods instead of 4 would take us from a 2.56% chance of success to 34%. An AI agent could calculate the expected time costs of gathering another blaze rod, than it could compare the outcome of this time calculation with the returns found from the cumulative chance distributions in Table 3.1. With this data our AI agent can make an informed decision about whether to collect an extra blaze rod.

3.3.3 Ender pearls

The quickest way to acquire Ender pearls is to trade them for gold. You do these trades with an entity called "Piglin." In order to do one trade, we need one gold piece. A trade has a 20 in 424 chance of giving Ender pearls. A single trade returns anywhere from 4 to 8 Ender pearls, uniformly distributed.

Eyes	$f(x)$ $P[X = x]$	$F(x)$ P[X ≤ x]
0	0.0000	0.0000
1	0.0000	0.0000
2	0.0000	0.0000
3	0.0000	0.0000
4	0.0000	0.0000
5	0.0000	0.0001
6	0.0005	0.0005
7	0.0038	0.0043
8	0.0213	0.0256
9	0.0852	0.1109
10	0.2301	0.3410
11	0.3766	0.7176
12	0.2824	1.0000

Table 3.1: The precise returns of our binomial distribution with x as the amount of eyes in possession, and the matching cumulative binomial distribution. ($P[X \leq x]$).

We could calculate how much gold we need to have a chance of above 2.5% of getting enough Ender pearls. Instead we chose to perform a Monte Carlo simulation [Das20] to estimate these chances. In the Monte Carlo simulation, a trading session is considered successful if we receive eight Ender pearls or more. We found that trading only three gold is enough for a 3.23% success rate. As gold is mostly gathered in chunks of nine ingots, we will most likely have nine ingots to trade. This will give us an average chance of 12.22% of acquiring enough Ender pearls through trade. The chances of the success rates, on different amounts of gold, can be found in Figure 3.3.

3.4 Items for world traversal

To get to the Ender dragon with a world record pace, we will need two additional portals for fast travel. We call traveling "fast travel" if we purposely travel through the Nether to save time. Traveling through the Nether is faster, as one block traveled in the Nether equals eight traveled in the Overworld. Two portals require twenty obsidian. The fastest way of obtaining obsidian is trading as well. We can no longer use the obsidian-making method that we previously used, as we will need to build the portals in the Nether, where there is no water to make obsidian with.

As we have previously stated, trading for items requires gold, however, we do not necessarily need to gather more than nine gold (that we previously already needed to require enough Ender pearls,) as the returns that do not end in Ender pearls, could end in obsidian. The probabilities of obtaining obsidian and Ender pearls are not mutually exclusive.

As we have already performed a Monte Carlo simulation that checks if we have enough of one required item, we can alter the code to check for more items. This way we do not have to make calculations on complex mutually dependent return distributions. We have more items that we will need to trade for

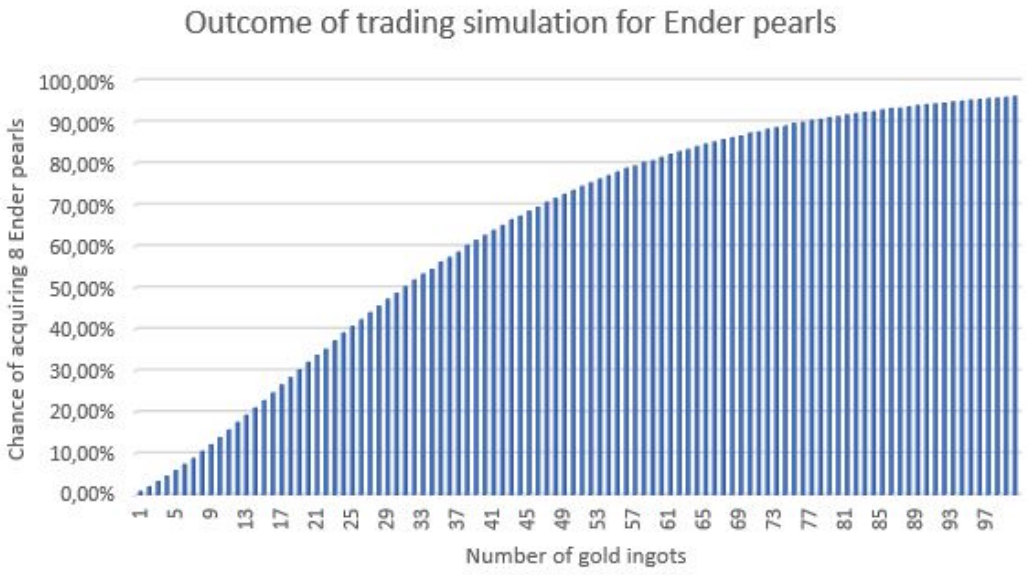


Figure 3.3: The percentage of successful trading sessions in the Monte Carlo simulation, We have a successful trading session when we encounter 8 or more Ender pearls in our returns.

later on in the dissection of the minimum required list. To avoid calculating this five separate times, we do them all in one simulation. You can find the code for this simulation in appendix A.
We used the following settings in function checkDesiredSuccess():

- desired_amount_pearl: 8
- desired_amount_string: 36
- desired_amount_obsidian: 22
- desired_amount_crying: 3
- desired_amount_glow: 3

The results of our simulation are in Figure 3.4. We can see that with 153 gold ingots we get an average success rate of 3.03%. This is just above the required 2.5% cut-off point that we set earlier on.

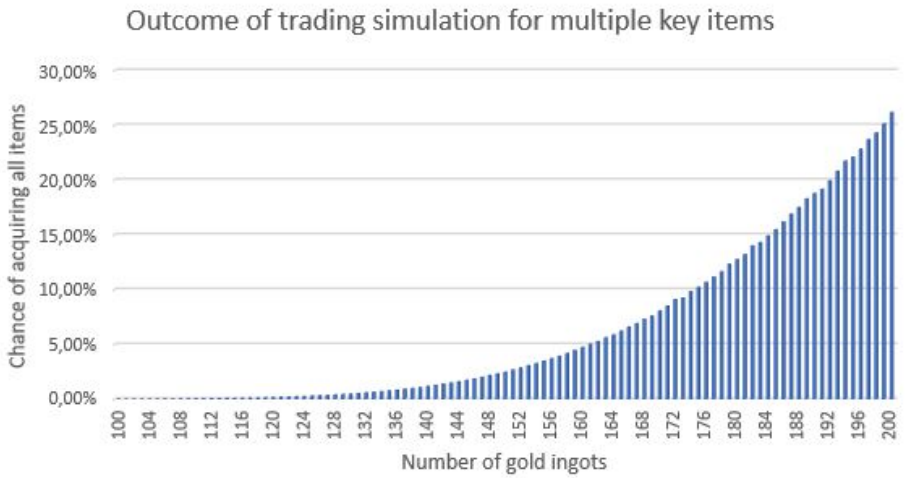


Figure 3.4: The percentage of successful trading sessions in the Monte Carlo simulation. We have a successful trading session when our returns contain sufficient quantities of all our key items.

3.5 Items for dragon killing

The fastest way to kill the Ender dragon is with explosives. Explosives in this case are beds and re-spawn anchors. Beds are made out of string and wood, re-spawn anchors are made out of crying obsidian and glow stone. They both have the same exact usage in the speedruns.

As an aside: One may ask themselves: "Why do beds explode?" Beds in Minecraft are used to bypass the nights in the game. This is possible in the Overworld (starting dimension), but there is no day-night cycle in other dimensions. This means the bed is useless in these dimensions, but the maker of Minecraft thought of something different. Instead of doing nothing when trying to sleep in these dimensions, the bed explodes. This often kills you.

The bed exploding gimmick is found out to be very time saving in speedruns, as the damage is calculated by the distance to any surrounding entity. When the dragon is close to the bed, the explosion will reduce its health by about a quarter. It requires two blocks of obsidian to put the bed close enough to the dragon, and subsequently far enough away from you. We specifically need obsidian as this material is so hard that it does not disappear after a blast. The obsidian will enable the player to quickly have all the beds explode, as he does not have to stop in between the explosions to replace the exploded blocks. This also brings our total minimum required obsidian from 20 to the 22 mentioned above in the Monte Carlo simulation.

3.6 Minimum items list and application

The previous paragraph concluded the description of the required items. This brings us to the following minimum item list:

- 0 building blocks (as it is theoretically possible to complete the game without any)
- 4 iron
- 4 logs of wood
- 1 flint
- 10 lava source blocks
- 17 gold blocks (153 gold)
- 0 food items (as it is theoretically possible to complete the game without any)
- 22 obsidian
- 3 crying obsidian
- 36 string
- 8 Ender pearls
- 4 blaze rods
- 3 glow stone

For each of these items, the AI agent should develop an utility function. The utility function assigns a value appreciation to each additional item gathered. The domain of the function begins with the minimum quantity required for the item. This is due to the fact that when we do not have the minimum amount required, we do not care about their cost, as not having them will make it statistically too unlikely to complete the game. We admit that it is hard to say exactly when chances are too unlikely, as we still have a couple of known unknowns. Further research in the likeliness of a speedrunning completion is needed here. The only way to truly research these factors is by letting the AI agent run a lot of times. Now, for demonstration purposes we will have to take these arbitrary chances (cut-off points at 2.5%), as these chances, when accumulated, seem likely enough to complete a run, while still having a fast time. We can see these starting points as hyper-parameters, which may be tweaked. However, in order to do so we first need an agent capable of doing the tests.

Of course the order in which the items are acquired will also be considered by the AI agent, depending on the state it finds itself in. For instance, when a required item is close and gatherable, the agent will most likely choose that task first, until its utility function for that item is satisfied enough to go search for other items on the list.

3.7 World record

The amount of gold needed in the previous section is way more than the gold used in the world record run. At first glance it seems likely that the world record holder cheated by altering the return distributions, as he only used 90 gold.

With our Monte Carlo simulation we can estimate how lucky the returns of the trades actually were, according to the Monte Carlo simulation. When using only 90 gold, the chance of finding these returns, or anything better, equals 0.00162%.

Recently, another speedrunner has been accused of cheating. Andrew Gelman calculated the statistical likelihood of him not cheating, given his extreme good fortune [Gel20]. After analyzing a series of his consecutive speedruns, the author of the paper concluded that the chances of these events happening were simply too low to be plausible. In Section 7 of the paper, he calculated the returns of Ender pearls and blaze rods using the same binomial formulas as we did. He concluded that the naive estimate of the likelihood of these events occurring was $4.97 \cdot 10^{-23}$. Due to the overwhelming evidence, the speedrunner had no choice but to admit to cheating.

Fortunately, it seems that our current world record holder did not cheat. The 1 in $6.2 \cdot 10^4$ occurrence is not that unlikely for a game played as much as Minecraft. This figure is comparable to figures found in other games.
The 90 gold used in the run is also misleading because the player already found some obsidian and string in a chest beforehand. When the AI agent does its speedrun attempts, we will need to account for these findings by shifting the return distributions in real time.

With the same simulation that we used in Section 3.4, we can calculate the likelihood of the world record holders trading returns.

The world record holder got the following returns from trading:

- pearl: 23
- string: 37
- obsidian: 7
- crying obsidian: 28
- glow stone: 50

The chances of finding these returns (or better,) according to the Monte Carlo simulation is a more manageable 0.213%. These are the chances at the tail of the distribution that you would expect in a world record run.

3.8 Chances for a world record

Throughout Chapter 3 we applied Monte Carlo simulations to estimate probability distributions, where we accepted probabilities that were just above the cut-off point of 2.5%. For instance, we accepted that the probability of acquiring the minimum number of eyes of Ender through trading to be 2.56%. When attempting to beat the world record, we will have to make a trade-off between success rate and time investment. The reason we have to work with such a low success rate is because a run will need to involve a certain amount of luck to have a chance of being a world record. To illustrate this, imagine

that we always receive the exact same world, and we use the same deterministic agent that tries to finish the game. The only difference is the amount of eyes of Ender that the agent is instructed to collect. The agent that is instructed to gather 8 eyes of Ender, will be slightly faster at the portal than the agent that is instructed to gather 9. When these agents play the game in this world a lot of times, on average the agent with 9 eyes will complete it in 11.09% of the runs against 2.56% of the agent with 8 eyes, as seen in Table 3.1. The fastest run, however, will belong to the agent with 8 eyes of Ender.

When we take the product of the two main luck factors, which are the returns of the blaze rod collecting and the trading, which are independent of each other, we get 2.56% × 3.03%. This equals a chance of 0.07%, which seems small, but encounterable when playing a large number of games. There still is one remaining chance factor: the world's layout.

The probability of a world having a good layout for a Minecraft speedrun is hard to determine, there is no literature on this. To calculate this, we would need the total number of world record attempts, but there are no logs of these attempts, as only the good attempts are noted. Using only the accessible runs will result in a massive bias, which we cannot adequately compensate for.

To make the best feasible prediction at this time, we may examine top speedrunner methods once again. Most speedrunners start up nine games of Minecraft simultaneously. They do this in order to reduce the time it costs them to find a world that looks promising for a world record. They often choose one of these nine worlds to play out. The number nine is pretty arbitrary. This seems to be a favorite choice, as nine adjustable rectangles tile a rectangular display nicely. Also, nine simultaneous games appear to be close to the maximum computing speed of a high-end desktop. It might also be hard to divide your attention on more than nine worlds at the same time, without losing too much time on your run.

We can take this speedrunning habit of 1 in 9 as an average for our chances of finding a good world. We admit that this remains somewhat arbitrary. To determine a better estimate for the probability of a Minecraft world that is speedrun-able, we would need a form of a speedrunning agent, to run out a large number of worlds. When using the one in nine chance we will come to a 0.008% chance of success on a speedrun. With this estimate we would expect a world record on perfect play to occur once in 12,500 games. With the current world record at ten minutes, playing 12,500 games will take a maximum of 125,000 minutes, as we can simply quit playing if we pass the record time. Playing this much for a human will take around 261 days if you work eight hours a day and you have speedrunning Minecraft as a profession. This may be a plausible situation for the majority of individuals who have held a world record at some point, which indicates that our predicted probabilities are not far off.

3.9 The costs of a world record

It will take an individual a long time to set a world record, as one can only play one game of Minecraft at a time. However, an agent playing Minecraft is not bound by one instance of Minecraft, as the agent could run in parallel. We could run this agent playing Minecraft on Google's cloud services. There are already instances of Minecraft running on Google's App Engine. The only alteration we would need to make in setting up an agent playing Minecraft on the cloud, would be to compile the game with the

agent code injected into it. A game of Minecraft requires 2 GB of RAM to run. The pathfinding agent we created for this thesis does not need to allocate extra RAM. This is however susceptible to change as the entire speedrunning AI agent, which includes the pathfinding agent, will be computationally more expensive. In the cloud you are mostly billed by the RAM you use over a certain amount of time. 2 GB of RAM costs 0.00002900 dollars per second. For 125,000 minutes of run time this will cost us around 217.5 dollars according to Google [Clo22].

We recently performed a cloud computation that cost roughly the same amount. When calculating the amount of RAM used, and compensating for the amount of time the function was running in the cloud, we can conclude that there is little overhead cost with the current Google cloud functions. The experiment we conducted estimates the cost of Minecraft running for 125,000 minutes in the cloud to be approximately $225.

If our approximation for Minecraft worlds having a correct layout for speedrunning is not magnitudes off, we can conclude that computing power and costs should not be an issue when running AI agents to compete with humans for the Minecraft speedrunning world record.

Chapter 4

Methods

Minecraft's ultimate goal is to defeat the Ender dragon. In Chapter 3 we divided this goal into subgoals. These subgoals are mostly finding materials, which are found in certain locations. To optimize the order and place of these tasks effectively, we need to know the time it will take to get from one location to another. This is why optimizing traversal in the game of Minecraft is an essential first step for an optimal player, as the rest of the AI agent to finish the game will build on this foundational computation.

A popular algorithm for world traversal with clear distinct states is the A* algorithm [HNR68]. Minecraft has clear distinct states: the blocks that the world consists of. Minecraft enables the player to walk in a continuous space, which is unusual for the A* algorithm. For this reason, we need to modify the standard A* algorithm in order to apply it to the world of Minecraft.

4.1 A*

The A* algorithm, which was first described in [HNR68], is a variation on Dijkstra's algorithm. Dijkstra's algorithm uses best-first search to find the optimal path. We also do this in A*. In A* euclidean distance calculations are utilized in order to reduce the complexity of the algorithm. This means that the problem must involve state locations with meaningful distances between them, which is not always the case in graph theory. These distance calculations give A* the advantage that it does not have to look at a node that can not theoretically be the fastest path, by the reasoning that even in the best case scenario, it will still be a longer path than our current shortest path. Even though the A* algorithm was originally published in 1968, it is still widely used today in well known applications like Google maps [MKL19]. The pseudo code for the A* algorithm can be found in Algorithm 1.

Algorithm 1 A* algorithm

procedure A*(v_c = current state, v_s = succesor state, E= Goal state, $d()$ = cost of getting from v_c to v_s, $h()$ = heuristic distance to goal, $g()$ = cost of current shortest path to get to v_s, $f() = g() + h()$ = current best case scenario cost of getting to E)

Set the open and closed list to an empty list and add the start node to the open list

openList $\leftarrow \emptyset$
closedList $\leftarrow \emptyset$
openList.$add(v_1)$
$f(v_1) = 0$
while openList $\neq \emptyset$ **do**
  $v_c \leftarrow min(f(V))$
  closedList.$add(v_c)$
  openList.$remove(v_c)$
  if $v_c = v_g$ **then**
    break; ▷ Solution is found, goal node is current node
  end if
  $V_S \leftarrow successors(v_c)$
  for $v_s \in V_S$ **do**
    if $v_s \in$ closedList **then**
      continue;
    end if
    $g(v_s) \leftarrow g(v_c) + d(v_s, v_c)$
    $h(v_s) \leftarrow d(v_s, E)$
    $f(v_s) \leftarrow g(v_s) + h(v_s)$
    if $v_s \in$ openList **then**
      if $g(v_s) > g$(openList) **then**
        continue;
      end if
    end if
    openList.$replace(v_s)$
  end for
end while
return openList

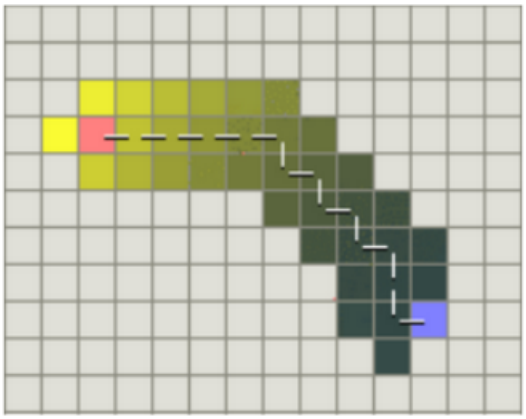


Figure 4.1: In this illustration we can see the A* algorithm and the path it selects in a grid world. All the colored squares are the nodes that the algorithm has considered at some point, as they were once adjacent to a state that has been explored further.
Source: Introduction to A* From Amit's Thoughts on Pathfindings.

4.2 A* in Minecraft

The A* algorithm is designed for state traversal. State traversal can be applied to a grid world by making every square a state. In a grid world one can normally only traverse states orthogonally, an example of this can be found in Figure 4.1. In Minecraft however, you can walk orthogonally and diagonally. The makers of the popular pathfinding bot: Baritone [Cab22] solved this problem "ad hoc" by turning diagonal Criss-Cross walking into a straight line wherever this is possible (the green line in Figure 4.2,) saving a lot of time in the process. Although this is a straightforward optimization, it will not always result in the optimal solution (the red line.)

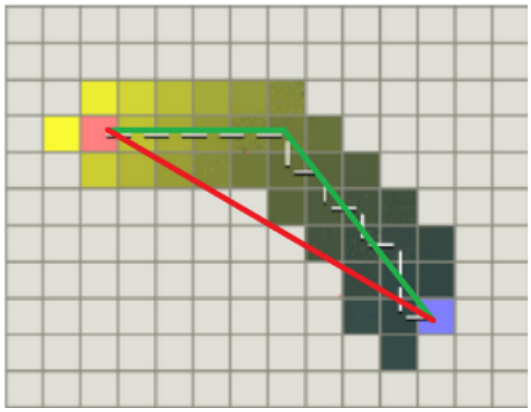


Figure 4.2: The optimal path (red) versus the path Baritone found (green.)

4.3 Proposed pathfinding algorithm: BlockStep

We refer to the algorithm that we developed as BlockStep. For path calculations the algorithm builds upon A*. Before the player starts to move, it calculates and saves this path to memory. The path consists of a series of blocks/states. When traversing the path, the bot moves from state to state with precise instructions, moving onto the subsequent state once its current goal is reached. Instructions may include specific jumping lift-off areas.

4.3.1 Jumping considerations

The timing of jumps is not straightforward, as the game runs at a 20 tick speed (20 game updates per second.) This means when we give the instruction to jump, we will jump anywhere from 0ms to 50ms after the command is given. It might be optimal in some cases to jump at the edge of a cliff to the other side, but because of this possible delay, we need to jump 50ms before reaching the edge. Otherwise we would have the chance of accidentally falling into the canyon. We calculated a safe lift-off zone between each state pair to solve this delay issue. This can be calculated by determining the last point from which we could theoretically jump to reach the succeeding block. This point is moved a specified distance towards the center of the current block. This distance will depend on how far we can travel in 50 milliseconds at the speed that we have when moving between those specific states. This works as we move from the center of each current block, to the center of the succeeding block. The agent component that traverses the path will now wait until we have passed this point before issuing the jumping command. In this manner, we will ensure that each jump can be executed safely.

Our forward momentum is 1.3 times greater when we are jumping compared to when we are solely sprinting. One may wonder, if jumping is so much faster, why are we not always jumping? This is because we cannot always utilize the full airtime of the jump. This occurs when we cannot jump the entire distance due to an obstacle or a gap, an example of this is shown in Figure 4.3. If we cannot jump the entire distance, we must slow down. The total length of a jump is four blocks, and it always takes the same amount of time from leaving the ground to landing again (when landing on the same level).

When jumping three blocks far, it will take us the same amount of time as jumping four blocks far.
The time it takes us to jump four blocks is 0.548 seconds, as the forward speed while jumping is 7.296 blocks per second (4 / 7.296 = 0.548). The time it will take us to jump three blocks far will be the exact same, as we will have the same amount of air time.

When we are sprinting three blocks without jumping, this will take us 0.535 seconds, as forward speed without jumping is 5.612 blocks per second (3 / 5.612 = 0.535).

Clearly 0.548 seconds is longer than 0.535. This time difference is small, but it does tell us that any jump equal to, or under three blocks is not worth it, if we can also walk that path. The times used in the calculations can be found on the Minecraft Wikipedia page [Wik22].

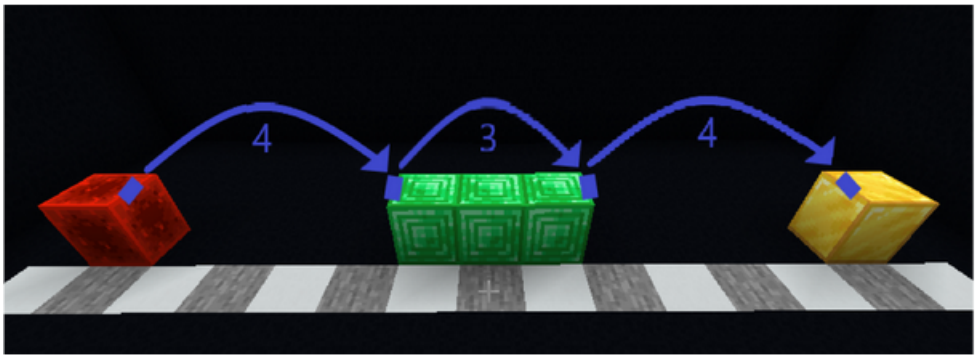


Figure 4.3: A situation where we can only jump 3 blocks in the middle jump. We should not take this middle jump, as walking this part (green blocks) will result in the fastest path. The blue squares on the edges of the blocks are the surface area of the player's feet at the time of jumping and landing.

When making small jumps, it takes the same amount of air time as large jumps. We now have remaining airtime that our algorithm must manage.
We do not simply stop moving when we arrive at our target block. We know before we jump how far the jump is, as BlockStep plans ahead in time. This allows us to smooth out the jump by gradually decreasing our speed mid jump. This results in a smooth parabolic jump, no matter the distance of the jump. Making all the jumps parabolic is very important for finding out what objects can be in our way when we calculate this in Section 4.3.3.
When determining which path is the quickest, BlockStep must account for these jumping speeds. When calculating the quickest route, jumping speed is crucial because it indicates how long it will take to reach each block in the path.

4.3.2 How many states are directly accessible from a given state?

BlockStep starts at the state/block on which the player is currently standing. BlockStep should now determine to what blocks the algorithm can go, directly from the current state.

In the literature we could not find to which succeeding blocks we could get to from the current block, while not touching another block. The calculations for this are not that straightforward, as we again have to keep in mind the restrictions of the update loop.

We can answer this question, as we have made an agent that can precisely jump reliably to where it wants to go. If our agent makes the jump, we know that this block is reachable as the agent we made already accounts for the 50ms delay we may encounter while trying to jump. We can divide this experiment into multiple case studies, for all the different height levels. the findings of this experiment are in the next subsections.

Ground level

First we look at the level on which the player is currently standing. We can jump a distance of four blocks. The blocks that are reachable when jumping at the same level are shown in Figure 4.4.

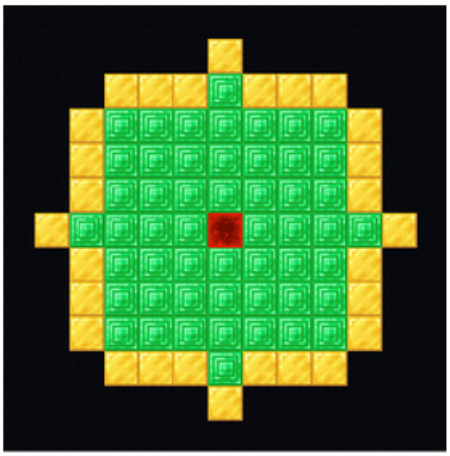


Figure 4.4: Reachable states (green and gold) when jumping at the same level from the starting state (red).

As we can jump four blocks far, we can manage to jump a four block gap (green blocks) and reach every gold block from the red one. We can manage to do this because the player has a square surface on its feet to connect with the world, as demonstrated in Figure 4.3. When part of its body is already in the gap, we can still jump as another part will still be on the block. When we land, the part of our body that was over the edge will connect to the block on the other side of the gap, resulting in a clearing of the jump. We can simply jump with less forward force to reach all the green squares. This brings the total number of reachable states for the same level to 80.

Going up one level

When jumping, we can get up 1 block higher than we currently are. With the parabolic nature of the jump, we can get to fewer blocks when ascending than we could when we remained at the same level. The blocks that are reachable when jumping up one level are shown in Figure 4.5.
The gold blocks are the furthest ones we can jump to, with the closer green ones also being jump-able to when these would be one level higher. There are some situations where we can not jump to the gold blocks, for instance when specific green blocks would be the same height as the gold blocks, they could be in the way of the parabolic jumping trajectory. There would need to be an extra check to see if no blocks are in the way of a player when moving from state to state. This check is explained in Section 4.3.3. When going up one level we can reach 68 states.

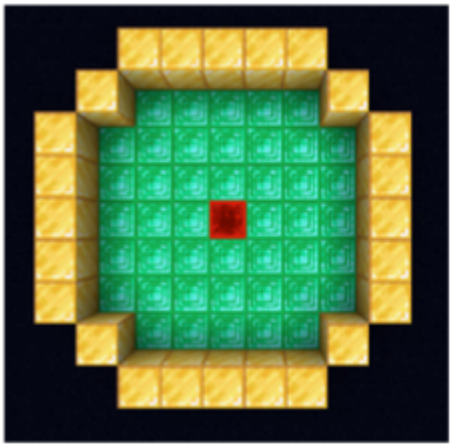


Figure 4.5: All the states we can reach when jumping up one block starting from the red block in the middle.

Going down multiple levels

We cannot ascend more than one block due to our limited maximum vertical jumping force, but we can descend a considerable distance due to the assistance of gravity. We do get a clear cutoff point for our algorithm, as falling more than three blocks will result in fall damage. When our next state is three blocks down, we can not jump, as this will result in us falling more than three blocks. With the speed increase we get when we jump, we can reach more blocks when jumping off a two high ledge, than simply walking off a three high one. The blocks that are reachable when jumping from a one or two block high ledge are shown in Figure 4.6.

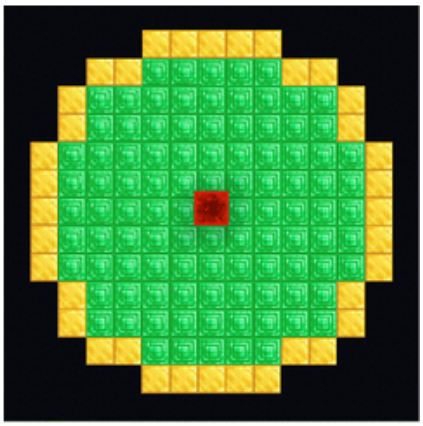


Figure 4.6: All the states we can reach when jumping from a one or two block high ledge(red).

when jumping from a one or two block high ledge, we can, for each distinct height, reach 136 possible states.

This brings the total number of states that are directly reachable from a given state, while not taking fall damage in Minecraft, to 420.

For every state that we explore in our path finding algorithm, we consider its 420 potential neighbors. We cannot, however, get to every block, as we can only access a block, if there is at least a gap of height 2 above it. BlockStep therefore first checks if the block we are trying to get to has a 2 height gap above it, before adding this block to the potential successor list.

Having 420 states that may be reachable from a given state in one step, implies that there are 420^k potential k-steps paths from the same state. This is a lot. Fortunately, with A* we do not have to explore all these states and all these paths. Firstly, as explained above, the actual number of reachable neighbours of a given state is much less than 420. Secondly, many other states will not be explored further as they are too far out of the way to result in an optimal path.

4.3.3 Obstacles in trajectory

Travelling from one state to another can be hindered by blocks in our way. We need to device a method to determine if there is no block in the way when going from one state to another. When we walk in a straight line, this is simple. As an example, we may wish to walk from coordinate z = 5 to z = 8, as shown in Figure 4.7a (In Minecraft, the x coordinate is north to south, the y coordinate is up to down, and the z coordinate is east to west.) Now we only have to check if there is a path available in the coordinates z = 6 and z = 7. This is possible if there is a 2 height gap at both coordinates.

(a) A situation where we can not get to the green block (z = 8) from the red block (z = 5), as a block is in our way at eye level.

(b) A situation where we can not get to the green block(z = 5) from the red block(z = 8), as a block (gold) will be in our way when we jump.

Figure 4.7: The following Figures showcase the blocks that could be in our way when traversing states.

When we jump, we have a parabolic trajectory. This makes the calculations of what blocks can be at which location more complex. A jump will bring our feet above the height of one block for part of the trajectory, meaning that some blocks are all right being in our way on a plus one level, while in the previous no jump example they were not. The jump also brings our head up one level, so for that part of the trajectory we need to have a three high gap instead of the standard two high gap. In Figure 4.7b

we see an example of a block that can be in our way when we jump. The trajectory also changes as we jump at different speeds and to different heights, as explained in Sections 3.3.1 and 3.3.2. In the example found in Figure 4.7b we find the case in which BlockStep would like to examine a jump in a zero degree east direction, but we have 420 different possible jumps, all having unique angles.
We suggest two ways to address this issue:

1. We can simulate each of the 420 jumps that BlockStep considers to take, for every possible combination of present blocks and absent blocks on our way. If the bot gets to the target block consistently when jumping, we know for sure that we can make the jump. We need to make sure that the bot clears the jump multiple times in the same environment, as the shifting of the lift-off point could give us inconsistent findings. When we are sure that we can either make, or not make the jump, we can save this outcome. Now we can later look up the specific jump given the specific environment. The training time for this would take a long time, as there exist 420 possible jumping situations with each, a lot of blocks that could either be present or absent.

2. We can also simulate our trajectory on the run in order to see if we touch any blocks. This means that we need to sample multiple points in the trajectory, in order to see if we touch a block other than our goal block. The number of points that we need to sample is high since the player has a continuous state space, which is only capped by the rounding of a floating point in the programming language Java. This would be an extremely expensive calculation to do for every block that we consider in the A* algorithm.

We chose the second option as there is a way to bring down its complexity. This way consists of increasing the margin around the player. This means that if we find our trajectory to be close enough to a block, we simply say that the jump is too risky. To further increase the precision while keeping the complexity low, we could add a ray marching algorithm, which is often used in trajectory hit detection [Bre22].

In order to determine the size of the margin, we conducted several tests in which we weighed the reduction in calculation time against the increase in path travel time. One tenth of a block of space did not significantly slow down BlockStep. Since our goal is to optimize travel time, we opted for this choice, rather than one that reduces calculation time more.

In principle BlockStep jumps all the time, as this is the fastest method of traveling. There are a few instances in which BlockStep calculates that jumping is not the optimal solution. These instances include:

1. As we have discussed in Section 4.3.1, BlockStep does not jump if it can not make a jump larger than three blocks across, which is common when we have steep inclines in the terrain.

2. BlockStep does not jump when there is a ceiling above his fastest found path, like we see in Figure 4.7b. Situations like this frequently occur in dense forests, as the leaves of the trees are often just high enough to walk under, but not high enough to jump under.

3. BlockStep does not jump when this is advantageous for the rest of his path. We can see in Section 4.3.5 that BlockStep sometimes moves relatively slowly to a certain spot, but from this spot, it will have a straight line to its goal, where it can make a long series of optimal jumps.

4.3.4 How to prevent algorithms that can read world data from cheating

BlockStep depends on having access to all the knowledge in the Minecraft world. A player on the other hand,can not calculate his complete path in advance, as he does not get to see the whole world. There is a variety of ways in which we could modify BlockStep to play fairly. For example, we could use statistics on world generation features to make a likelihood function of finding a fast path in a given area that we can not yet see. We can then move toward this area. Once we can fully see the area that we need to traverse, we can switch back to our optimal A* algorithm.

Since we are comparing BlockStep to Baritone, which uses information from the entire world, it would only be fair if we do the same, although it would be an even greater accomplishment to beat Baritone without cheating, which I believe is possible. This would require a series of much more complicated algorithms. A reinforcement learning algorithm that would automatically learn the terrain characteristics seems to be a suited solution here.

4.3.5 BlockStep in action

When we combine all the steps described above, we get an algorithm that is optimized for path traversal in Minecraft. We can now see if it performs like we expect it to. In the video linked in the footnote[1], the goal given to BlockStep is to get to the tower. We can see that BlockStep is smart, as it chooses not to walk directly towards the tower, as it finds a small hill in the way. BlockStep first moves a little to the left of the hill in order to be able to make faster jumps. At twelve seconds into the video, we can see BlockStep making two small slow jumps to the right. This again seemed to be a pretty good move, as BlockStep then had a straight path where it could again jump optimally towards the tower.

4.3.6 The code-base of BlockStep

The GitHub link of the complete environment of BlockStep can be found in the footnote[2]. The environment is a decompiled version of Minecraft 1.16.1. This version of Minecraft has very little support or tutorials for programming. There are versions with a lot more support like 1.16.5. This is probably the reason why Baritone and AltoClef were built on that version, even though the goal of AltoClef is speedrunning Minecraft, and as established before, speedrunning Minecraft is done best in version 1.16.1.

[1] A video of BlockStep in action

[2] The code base of BlockStep

Chapter 5

Results

5.1 Baritone

Baritone is a widely used module to automate processes in Minecraft. The most prominent feature of Baritone is its ability to find and execute a path to a given destination.

As Baritone has the best pathfinding algorithm in Minecraft to date, it is a good benchmark to test BlockStep against. Baritone was first released in August 2018, has 54 contributors, and is still being updated.

If we want to test both systems extensively, we must automate the test and have a system logging the travel times. We have full control over our own code, which makes it easy to log all the statistics there, however to get Baritone benchmark statistics we need to install a complete test street. Our Baritone test street can be found in Appendix B.

5.2 Benchmarking BlockStep

Minecraft has all kinds of different biomes, all with their own terrain features. Some biomes will be harder to traverse for some algorithms than for other algorithms. This is why we tested the algorithms in multiple Minecraft biomes. In particular, we tested the algorithm in the following biomes:

Deserts, Hills with Trees, Ice, Ice Mountains, Swamps, and Trees.

In each biome, we select the middle point as a starting point. From the starting point, the algorithms will get a goal position to go to. The goal will be in either the North-West, North-East, South-West, or South-East direction. We have chosen these oblique directions, as orthogonal paths are less interesting to investigate. We chose for the end points to be 20, 50 and 100 blocks away from the starting point in both x and z directions. This means that the minimum distance of each path (euclidean distance) will be 28.2, 70.7 or 141,4 blocks long. The Baritone agent as well as the BlockStep agent started at the same location, and were given the same goal. For every biome, we tested four directions, each for the 20, 50 and 100 block distances. This resulted in a total of 72 tests per algorithm. In Table 5.1 we can see the results of these tests.

Test description	AVG traveltime Blockstep in Sec	AVG traveltime Baritone in Sec	Diff. in Sec	Max traveltime Blockstep in Sec	Max traveltime Baritone in Sec	Diff. in Max	StDev traveltime Blockstep in Sec	StDev traveltime Baritone in Sec
Move 20 20	**6.6**	**8.4**	**-1.8**	**14.6**	**13.5**	**0.9**	**2.3**	**2.5**
Desert	5.1	6.0	-0.8	6.2	6.1	0.1	0.8	0.1
Hills and Trees	7.2	8.4	-1.2	9.5	9.7	-0.3	1.7	1.3
Ice	4.7	6.6	-1.9	5.3	8.0	-2.7	0.4	1.2
Ice Mountain	7.6	8.0	-0.3	14.5	12.0	2.5	4.6	2.7
Swamp	7.8	11.1	-3.3	9.8	13.5	-3.7	1.4	1.8
Trees	7.3	10.6	-3.3	8.5	13.0	-4.5	0.9	2.5
50 50	**16.2**	**18.8**	**-2.6**	**26.0**	**31.8**	**-5.8**	**4.3**	**4.2**
Desert	12.0	14.6	-2.7	13.3	14.9	-1.7	1.0	0.3
Hills and Trees	17.6	17.8	-0.2	19.2	19.2	0.0	1.4	1.4
Ice	13.4	15.7	-2.3	18.3	17.4	0.9	3.4	1.4
Ice Mountain	20.2	22.1	-1.9	26.0	28.5	-2.5	6.0	4.5
Swamp	17.8	23.2	-5.4	23.1	31.8	-8.7	3.7	5.8
Trees	16.1	19.2	-3.1	21.6	20.4	1.2	3.9	1.0
100 100	**35.1**	**37.0**	**-1.9**	**49.3**	**45.7**	**3.6**	**9.5**	**4.5**

Table 5.1: Test results of Baritone versus the new algorithm: BlockStep. The times shown here are an average of walking in all four directions.

The results of the tests show that on average, BlockStep was faster than the Baritone on distances “20 20” and “50 50.” Most of the time, the maximum travel time of BlockStep was lower than Baritone. On the distances tested, BlockStep was faster 85.4% of the time. The standard deviations of the travel times are comparable, which means that for most of the paths, BlockStep will be faster. With the distance “100 100” we see that the algorithm scores are close to each other. The tests done on this distance were more challenging. This is because both algorithms were sometimes stuck. We can only compare the runs on which both algorithms did finish. Due to the survivorship bias found in the "100 100" distance results, these findings cannot be generalized.
BlockStep is a direct improvement over Baritone, as it investigates more neighbouring states, but BlockStep does not find a faster path in all cases, this is because in some instances both algorithms will find the same optimal path. In this case it is just a matter of which algorithm is more lucky with the game loop updates.

Another way in which Baritone can be faster is if we encounter a block with a shape that has not been accounted for in BlockStep. We assume in our algorithm that every block is either solid, liquid, or air. Even though more than 99% of the blocks fall into these categories, we still occasionally encounter some blocks that are not in this set. BlockStep has a fail safe mechanism for handling these abnormal blocks, but this mechanism is “ad hoc” and far from optimal. An example of one of these blocks is snow, which is not a full block in it’s height dimension. This will cause problems for the algorithm’s jumping predictions, causing it to miss a few jumps here and there. This is partly the reason that we did not see a lot of improvements in the Ice Mountain biome.

BlockStep encountered a problem when calculating a path for the distance "100 100" because Java would occasionally run out of memory. This is because of how Java handles garbage collection. It should be possible to find a way around this. A workaround could involve requesting all the data necessary to

calculate a path, so that we can then send this information to a C++ program that would calculate the path. In C++ we have the ability to assign and free our own memory. This will allow us to release the no longer-needed state objects from memory.

The test statistics demonstrate that BlockStep outperforms Baritone. Does this mean that BlockStep is superior to Baritone? In a sense, yes, because, on average, it discovers a faster path than Baritone. However, we did not allocate more processing power to the Minecraft environment in which BlockStep ran, nor did we fully optimize the algorithm, so the path calculation time for BlockStep is longer than that of Baritone's. This is expected given the years of development on Baritone with 54 contributors/programmers.

BlockStep did however demonstrate that there is a faster reliable path in Minecraft to be found than the path that the popular pathfinding algorithm Baritone finds.

Chapter 6

Conclusion and proposed optimizations

Breaking the Minecraft world record seems doable using an AI agent. an AI agent can already accomplish all the separate tasks needed to complete the game, as we have seen in Section 2.4. To combine all these separate tasks into one speedrunning AI agent, some work still needs to be done. In Section 3.6, we suggested a starting point in the form of an ordered list of items to be gathered, which the AI agent could be optimized on further. Once optimized on this list we will get a basic speedrunning AI agent, capable of finishing Minecraft in a fast manner while still having reasonable odds. We can parameterize these odds as soon as we gather the missing data. To gather this data, we would first need a basic speedrunning AI agent. In this thesis, we have optimized the path finding, which can be a foundational part of such an AI agent.

Baritone is the leading algorithm in the field of Minecraft path finding. Our pathfinding algorithm finds a faster path than Baritone in 85.4% of the test cases. BlockStep finds, on average a path that is 16% faster than the one that Baritone finds.
Our algorithm demonstrates that faster, more reliable paths can be discovered than those discovered by Baritone.
BlockStep's biggest advantage over Baritone is the fact that it checks every possible block it could theoretically get to in its pathing calculations, while Baritone only looks at eight blocks.

The path that BlockStep makes can in theory still be optimized, as we use whole blocks as states instead of the continuous space that a Minecraft player can move in. In practice, however, this may be hard as the safe jumping lift-off margins are close to the size of a whole block.

A possible improvement of BlockStep could be the inclusion of blocks that are not whole blocks. This optimization takes a long time as there are many different types of these semi-blocks. Moreover, it would not be a huge improvement as these blocks do not occur frequently in a Minecraft world.

Another optimization for BlockStep we could make is decreasing the calculation time. This could be by performing the same calculations in a more suitable programming language, like C++. To decrease computing time even further, we could run the A* calculations in parallel [Jor21]. This will, however, not further decrease the path traversal time, as in all cases we find the same path.

Bibliography

[Bre22] Anton Bredenbals. *Visualising Ray Marching in 3D*. PhD thesis, The University of Groningen, 2022.

[Cab22] Cabaletta. Baritone. https://github.com/cabaletta/baritone, 2022.

[CJhH02] Murray Campbell, A. Joseph Hoane Jr., and Feng hsiung Hsu. Deep blue. *ARTIFICIAL INTELLIGENCE*, 134:57–83, 2002.

[Clo22] Google Cloud. Cloud functions pricing. https://cloud.google.com/functions/pricing, 2022.

[Cub22] Cube1337x. Minecraft 1.16 speedrun world record. https://www.youtube.com/watch?v=dUW_huYo4cM, 2022.

[Das20] Dastilz. Minecraft bartering simulator. https://github.com/dastilz/mc-bartering-simulator, 2020.

[DHO12] David L. Dowe and José Hernández-Orallo. "IQ tests are not for machines, yet". *Intelligence*, 40(2):77–81, 2012.

[FWJ+22] Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. Minedojo: Building open-ended embodied agents with internet-scale knowledge. *Cornell University*, June 2022.

[Gel20] Andrew Gelman. Dream investigation results: Official report by the minecraft speedrunning team, 2020.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[Jor21] Nickson Joram. Parallel A* search on gpu. *Medium*, 2021.

[Mat22] Gaucho Matrero. Altoclef. https://github.com/gaucho-matrero/altoclef, 2022.

[M.F00] Robert M.French. The turing test: the first 50 years. *Trends in Cognitive Sciences*, 4:115–122, 2000.

[MKL19] Heeket Mehta, Pratik Kanani, and Priya Lande. Google maps. *International Journal of Computer Applications*, 178, 2019.

[Sha22] ShadowDraft. Minecraft java edition leaderboards. https://www.speedrun.com/mc, 2022.

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

[Skr22] Alexey Skrynnik. The MineRL competition. https://slideslive.at/38922880/the-minerl-competition, 2022.

[Vin19] Oriol Vinyals. Grandmaster level in starcraft II using multi-agent reinforcement learning. *Nature*, 575:350–3542, 2019.

[Wik] Minecraft Wiki. Noise generator. https://minecraft.fandom.com/wiki/Noise_generator.

[Wik22] Wikipedia. Minecraft. https://nl.wikipedia.org/wiki/Minecraft, 2022.

Appendices

Appendix A

Code to simulate the likeliness of getting specific key items when trading

```
//original code by github user: dastilz
//original can be found here: https://github.com/dastilz/mc-trading-simulator
//altered version by Isaac Braam
//this version has multiple desired items, and checks if we have all those items.

//There are many more item objects, but they are removed as
//these are not interesting for this paper.
//You can still find them in the original code.
let items = [
    {
        name: "Enchanted␣Book",
        min_quantity: 1,
        max_quantity: 1,
        drop_probability: 0.0118,
        current_drops: 0,
        average_drops: 0,
        total_drops: 0,
        desired_successes: 0,
        min_assigned_probability: 0,
        max_assigned_probability: 0
    }
 ]

function barter() {
    let random = getBarteringRandom()
    for(let i=0; i<items.length; i++) {
        let item = items[i]
        if (random >= item.min_assigned_probability
        && random < item.max_assigned_probability) {
```

```
            let drop_amount = getRandomInt(item.min_quantity, item.max_quantity)
            item.current_drops += drop_amount
            item.total_drops += drop_amount
        }
    }
}

function distributeProbabilities() {
    let max_probability = 1
    for (let i=0; i<items.length; i++) {
        let item = items[i]
        item.max_assigned_probability = max_probability
        item.min_assigned_probability = max_probability - item.drop_probability
        max_probability = item.min_assigned_probability
    }
}


function getRandomInt(min, max) {
    min = Math.ceil(min);
    max = Math.floor(max);
    return Math.floor(Math.random() * (max - min + 1)) + min;
}

function getItems() {
    return items
}

function getBarteringRandom() {
    let random = 0
    while (random == 0 || random == 1) {
        random = Math.random()
    }
    return random
}

function calculateAverageDrops(simulations) {
    for (let i=0; i<items.length; i++) {
        let item = items[i]
        item.average_drops = (item.total_drops / simulations).toFixed(2)
    }
}

function checkDesiredSuccess(desired_item, desired_amount) {
    //Ender pearl
```

```
    let pearl = items[8]
    let desired_amount_pearl = 23
    //string
    let string = items[9]
    let desired_amount_string = 37
    //obsidian
    let obsidian = items[14]
    let desired_amount_obsidian = 7
    //crying obsidian
    let crying = items[15]
    let desired_amount_crying = 28
    //glowstone
    let glow = items[6]
    let desired_amount_glow = 50

    if (pearl.current_drops >= desired_amount_pearl &&
    string.current_drops >= desired_amount_string &&
    obsidian.current_drops >= desired_amount_obsidian &&
    crying.current_drops >= desired_amount_crying &&
    glow.current_drops >= desired_amount_glow) {
        pearl.desired_successes += 1

    }

}

function getDesiredSuccesses(desired_item) {
    for (let i=0; i<items.length; i++) {
        let item = items[i]
        if (desired_item == item.name) {
            return item.desired_successes
        }
    }
    return undefined
}

function reset() {
    for (let i=0; i<items.length; i++) {
        let item = items[i]
        item.current_drops = 0
        item.average_drops = 0
        item.total_drops = 0
        item.desired_successes = 0
    }
}
```

```
function resetCurrentDrops() {
    for (let i=0; i<items.length; i++) {
        let item = items[i]
        item.current_drops = 0
    }
}

function runSimulations(simulations, gold, desired_item, desired_amount) {
    let probability = undefined
    let successes = undefined

    reset()
    distributeProbabilities()

    if (desired_item && desired_amount && desired_item != "Choose a desired item") {
        successes = 0
    }


    total_item_drops = 0

    for(let i=0; i<simulations; i++) {

        for(let j=0; j<gold; j++) {
            barter()
        }

        checkDesiredSuccess(desired_item, desired_amount)
        resetCurrentDrops()
    }

    calculateAverageDrops(simulations)
    successes = getDesiredSuccesses(desired_item)

    if (successes != undefined) {
        probability = {
            desired_item: desired_item,
            desired_amount: desired_amount,
            simulations: simulations,
            successes: successes,
            gold: gold,
            percentage: ((successes / simulations) * 100).toFixed(2)
        }
    }
```

```
    percentage= ((successes / simulations) * 100)
    console.log("percentage:")
    console.log(percentage)


    return {
        items: items,
        probability: probability
    }
}
runSimulations(10000000, 90, "Ender Pearl", 0);
```

Appendix B

Baritone's test street

Baritone's test street includes:

1. A python library called RaspberryJuice, which is a python API framework that we can use to request the position of the Minecraft user on a server, to observe the path traversed by Baritone in real time.

2. A server running a version of Minecraft that is compatible with Baritone as well as the RaspberryJuice python library.

3. A Python script that gives Baritone instructions on where to move to. The script does this by taking over the screen with PyAutoGui, and typing the commands in the Baritone interface.