Optimizing Minecraft Speedruns with Pathfinding Algorithm: A Heuristic Approach to Route Planning

Daniel Pedrosa Wu - 13523099
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: danielpedrosawu5705@gmail.com , 13523099@std.stei.itb.ac.id

***Abstract*— The competitive pursuit of Minecraft speedrunning represents a complex, high-dimensional optimization challenge, characterized by procedurally generated environments that demand dynamic route planning under severe time constraints. This paper investigates the application of computational intelligence to address this challenge. We propose a solution leveraging the A* search algorithm, a foundational technique in artificial intelligence for graph traversal, to automate and optimize navigational pathfinding within *Minecraft*'s three-dimensional voxel world. A software artifact, implemented as a mod for the *Minecraft* Fabric loader, serves as a proof-of-concept. This tool utilizes a Euclidean distance heuristic to calculate and visualize geometrically optimal paths between two points in real-time. The performance of this system is evaluated across a series of controlled in-game scenarios, measuring metrics such as calculation time, computational load, and path quality. The results demonstrate the viability of using heuristic-based search algorithms for route optimization in this domain. However, the analysis also highlights the inherent limitations of a purely geometric heuristic in the context of speedrunning, where temporal efficiency is paramount. This work establishes a baseline for computational assistance in *Minecraft* speedrunning and identifies critical areas for future research, including the development of more sophisticated, context-aware cost functions and dynamic re-planning strategies.**

Keywords—Minecraft, speedrun, sprint-jump, A*, heuristic

I. INTRODUCTION

Minecraft is a voxel-based sandbox game developed by Mojang Studios, has transcended its humble origins into becoming a global phenomenon. Initially created by Swedish video game programmer Markus “Notch” Persson and later sold to Microsoft in a multi-billion dollar deal, Minecraft has since became the best-selling video game of all time. The core gameplay, which revolves around the creation and exploration within its block-based platform serves as a platform for players to express their creativities and enjoy the game however they like.

One such activity that has been gaining popularity in the last couple of years is speedrunning. Speedrunning is the practice of completing a game, or a specific section in the shortest time possible, following the rules established for a specific category. In the context of Minecract, the most popular category is the “Any% Random Seed Glitchless (Any% RSG)” category, where the player’s objective is to defeat the final boss of the game, the Ender Dragon, starting from a new randomly generated world each run without the usage of game-breaking exploits.

Any% RSG speedruns are fundamentally different from other speedrunning categories or even other games. Unlike games with fixed level designs, the random nature of this category makes the run fundamentally unpredictable. This inherent randomness means that speedrunners cannot rely on memorizing a single, static route. Instead, speedrunners must adapt on the fly, relying on dynamic decision-making and rapid environmental assessment to navigate the complex, three-dimensional space.

However, the sheer scale and complexity of a Minecraft world presents a search space that is too vast for exhaustive human analysis. This means that the route a human player takes will not always be the optimal path. This makes the problem of route planning an ideal subject for computational augmentation. By modelling the Minecraft world as a graph and applying certain pathfinding algorithms, it is possible to mathematically calculate optimal paths between different points, potentially exceeding the efficiency of routes derived from human intuition.

This paper aims to design, implement and evaluate an algorithm capable of computing the geometrically shortest path within a complex, three-dimensional voxel world of Minecraft. To serve as a proof-of-concept, a mod was developed for the Fabric Mod Loader. This mod leverages the A* search algorithm, to calculate the optimal route between two coordinates specified by the user. The pathfinding process operates under the constraint that the path taken will result in the player taking minimal damage and that it assumes the player performs optimal movements. This path is then rendered visually in the Minecraft world through the usage of the game particle system. This paper provides a comprehensive account of this system, from its theoretical underpinnings to its practical implementation.

II. Theoretical Foundation

A. *Minecraft as A Sandbox Game*

Fig. 2.1. Minecraft
Source: https://minecraft.fandom.com/wiki/Minecraft

Minecraft is known as a “sandbox” game, a genre defined by the freedom it affords players to interact with the game how they see fit. This genre is characterized by the lack of a predetermined narrative or a linear set of objectives. The core gameplay loop of Minecraft consists of players exploring the procedurally generated world, gathering resources by breaking blocks or killing mobs and then using those resources to craft items and build structures.

There are two primary gamemodes in Minecraft which are Survival mode and Creative mode. In Survival mode, players must manage their health and hunger bars, gather resources manually and defend themselves against hostile creatures. In constrast, Creative mode removes all survival aspect. The players are immortal, have access to an unlimited supply of all items and gained the ability to fly, allowing them to focus purely on their creative expression.

The true driving force behind Minecraft’s popularity and longetivity is its vast and creative community. This global community of players extends the game beyond its original scope by creating a near-limitless amount of user-generated content. Such examples are mods which allows modification to the game’s code, maps which offers curated adventure or puzzle for other players to play. Thousands of multiplayer servers also create unique social network of players, cementing Minecraft as not just a game, but a platform whose limits are defined only by the collective imagination of its community.

B. *Beating Minecraft*

While Minecraft is renowned for its open-ended sandbox nature, allowing players to set their own goals, it does feature an implicit main quest line that provides a definitive "end" to the game. This objective-driven path culminates in a confrontation with the game's final boss, the Ender Dragon. Completing this goal requires players to navigate through the game's three dimensions—the Overworld, the Nether, and The End—and accomplish a series of specific tasks, which are:

1) *Entering The Nether*

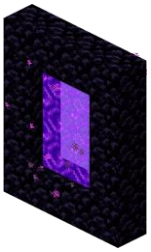

Fig. 2.2. Nether Portal
Source: https://minecraft.wiki/w/Nether_portal

The player’s journey begins in the Overworld. The player must gather the resources required to enter the Nether. The Nether is a hell-like dimension filled with dangerous creatures, imposing structures and unique resources. To enter the nether, the player must activate a Nether Portal. The portal is made by constructing an obsidian border around a rectangular area of empty space with a minimum internal dimension of 2 blocks wide by 3 blocks tall. This portal can then be activated by lighting the obsidian with a Flint and Steel.

2) *Acquiring Key Items in the Nether*

Fig. 2.3. Blaze and Enderman
Source: https://minecraft.wiki/w/Blaze dan https://minecraft.wiki/w/Enderman

It was mentioned beforehand that the Nether contains dangerous creatures and unique resources. One such mob is known as the Blaze, which drops an item called a Blaze Rod. Once entering the nether, the player must locate a specific structure known as a Nether Fortress. This structure is important because it is the only known place where the Blaze spawns. These rods are then used to craft Blaze Powder, which are part of the recipe to craft the Eye of Ender. Beside the Blaze Powder, the player also needs the Ender Pearl which are dropped by Enderman, a mob that spawns in every dimension.

3) *Locating The Stronghold*

Fig. 2.4. Eye of Ender’s Recipe
Source: https://minecraft.wiki/w/Eye_of_Ender

The Eye of Ender can then be thrown in the Overworld. When thrown, these eyes will fly towards the location of a hidden, underground structure known as the Stronghold. It is also important to note that travelling one block in the Nether is equivalent to travelling eight blocks in the Overworld which can vastly help with traversal. Once in the Stronghold, the player will have to find a specific chamber known as the Portal Room.

4) Activating the End Portal

Fig. 2.5. End Portal
Source: https://minecraft.wiki/w/End_portal

Once located, the portal room will contains End Portal Frame blocks which need to be filled by Eye of Ender to lit up. In total, there are twelve End Portal Frame and each portal room generates with a random amount of portal frames already filled in. Once all twelve are filled, the portal will then lit up and the player can enter The End.

5) Fighting the Dragon

Fig. 2.6. The Ender Dragon
Source: https://minecraft.wiki/w/Ender_Dragon

Upon entering The End, the player faces the Ender Dragon. In the arena, there are several obsidian pillars, each housing an End Crystal. These crystal allows the dragon to heal itself. Once the Ender Dragon is defeated, a portal that leads to the Overworld will open up, upon which the player can enter to return to the Overworld.

C. The Minecraft Environment as a Search Space

Every Minecraft world is generated using a pseudo-random algorithm determined by a numerical "seed". This seed is a 64-bit integer which dictates the entire structure of the world. This process is deterministic which means using the same seed in the same game version will always produce the exact same world. The world is composed of chunks which are 16×16 columns extending through the vertical height of the map. Chunks are generated on-the-fly, loading a small portion of the world to the players at any given time. This approach is used to create the illusion of an infinite world while keeping memory usage to a manageable level.

The Minecraft world is a discrete, three-dimensional grid composed of blocks. Every element in the world occupies a specific coordinate (x, y, z) in this grid. This discrete representation is key to modelling the Minecraft environment for pathfinding. The world can be abstracted into a massive graph where each block that the player can stand on represents a node, whilst movement between adjancent, traversable blocks represents an edge. The cost of traversing the edges of this graph is determined by Minecraft's movement physics.

In Minecraft, there are three primary modes of ground movement: walking, sprinting and sneaking. Each has a distinct base value applied to the player's velocity each game tick (every 0.05 seconds). Without external modifiers, the default walking speed is 4.317 blocks/second [3]. While sprinting, the player moves at a speed of 5.612 blocks/second [3] and while sneaking, the player moves at a speed of 1.3 blocks/second. This value can then modified by several factors, such as environmental factors (i.e. being airborne, being in water, taking damage, etc), status effects (i.e. buffs/debuffs) or the physical properties of the blocks. For example, sprinting while jumping yields an average speed of 7.127 blocks/second [3].

The game's physics model uses an absolute coordinate position which means that the player position and motion are defined relative to a fixed world origin, located at the coordinate $(0,0)$ on the horizontal XZ-plane. The three different movement modes and their associated speed can directly translate into variable edge weights in the graph. This means a sprint-jump—the act of jumping while sprinting—would have a lower time cost compared to walking or even sneaking the same distance.

D. Minecraft Any% Random Seed Glitchless Speedrunning

A Minecraft Any% RSG speedrun has a single objective: defeating the Ender Dragon within the shortest time possible, without the usage of game-breaking exploits. Although the game has evolved overtime, the key stages of an Any% RSG speedrun have largely remained the same, which are the Overworld Setup, the Nether Entry, the Nether Phase, the Stronghold Location and finally The End. While these foundational stages are consistent, the strategies within them have been highly optimized in the modern era of speedrunning. A typical run in the modern version of the game (1.16+) progresses through each of the stages as follows:

1) Overworld Setup

The run begins with gathering basic resources needed to craft essential tools. Since each run is on a different world, there are several ways that a run can begin. The immediate goal is often to locate a structure that provides a significant resource advantage, such as a village, a shipwreck, a buried treasure or a ruined portal.

2) Nether Entry

The player must then enter the Nether dimension. To do this, they must construct a Nether portal. To save time, speedrunners rarely mine obsidians directly. Instead, they use a bucket of water and a pocket of lava to directly create obsidians. This can be done on a lava pool or a lava ravine underwater. Alternatively, the player can also complete a ruined portal.

3) The Nether Traversal

This is the most critical stage of the run. There are two key resources from two different structures that the player must gather, which are the Ender Pearls and the Blaze Rods. Although

they are dropped from Endermen, the most consistent and fastest way to obtain them is through bartering gold with Piglins. Each barter takes about 8 seconds, however more than one Piglin can trade at the same time. A mass amount of Piglins are located in Bastion Remnants. These structures contain golds that can be used to trade with the Piglins.

The Piglins can also give other useful resources such as strings to craft Beds (explosives while in the Nether or The End), obsidians (to build portals) and potions. The next structure is the Nether Fortress. This structure contains the Blaze which drops the Blaze Rod. Usually a runner will have already acquired a Fire Resistance Potion which makes them immune to the Blaze's attack while bartering with the Piglins.

4) Stronghold Location

After acquiring a sufficient amount of Blaze Rods and Ender Pearls, the player can then return to the Overworld to locate the Stronghold. By throwing the Eye of Ender which can be crafted using the resources gathered, the player can know the direction of the Stronghold. Since travelling in the Nether is 8 times faster than in the Overworld, the player can predict where the Stronghold will be and travel to the predicted coordinate from the Nether. By constructing a portal there, the player can then end up inside the Stronghold.

5) The End

Once in the Stronghold, the player must locate the portal room, activate the portal with the Eyes of Ender and enter The End dimension. The final challenge is to beat the Ender Dragon. Instead of using traditional weaponry such as swords or bows, speedrunners opt to use beds to kill the dragon as they explode when used in the Nether or The End.

E. The A Search Algorithm for 3D Route Planning*

1) Formal Definitions

As presented beforehand, a lot of travelling is involved in Minecraft speedrunning. Finding the most efficient path between two different points is the problem that this paper is trying to solve. The A* search algorithm is an informed, best-first search algorithm that operates on a weighted graph to find the path of least cost starting from the start node to the goal node [2]. At each step, it will evaluate a node based on the evaluation function $f(n)$:

$$f(n) = g(n) + h(n).$$

where:

- $g(n)$ is the cost function, which represents the known cost of the path from the start node to the current node n.
- $h(n)$ is the heuristic function, which estimates the cost of the cheapest path from the current node n to the goal node.

The A* algorithm maintains two lists of nodes which are:

- Open List: A priority queue containing all nodes that have been discovered but not yet fully evaluated or expanded. The nodes are ordered using their $f(n)$ value, with the lowest-value node being the highest priority.
- Closed List: A set containing all nodes that have already been evaluated to avoid redundant computation.

The algorithm works as follows:

- Initialize the Open List with the start node s with $g(s) = 0$ and $f(s) = h(s)$.
- Until the goal node is selected for evaluation or the Open List is empty, the algorithm proceeds to iteratively:
    - Removes the node with the lowest $f(n)$ value from the Open List and add it to the Closed List.
    - Evaluate its neighbors, check if they are in the Closed List and update their costs if a better path is found then adds them to the Open Set.

2) Heuristic Function

To guarantee that A* finds an optimal solution if it exists, the heuristic function $h(n)$ must have admissible. A heuristic $h(n)$ is admissible if it never overestimates the true cost of getting from the current node n to the goal node [2]. This means that for every node n, the value of $h(n)$ must be less than or equal to the actual cost from the start node to the goal node. Mathematically this can be expressed by:

$$\forall n \in N, h(n) \leq h^*(n)$$

By never overestimating, the heuristic ensures that A8 doesn't prematurely discard a node that on the surface seems inefficient. A* will always prioritize the path with the lowest $f(n)$, which means that $f(n) = g(n) + h(n)$ needs to act as an optimistic best-case scenario for a path going through n to ensure that a path doesn't get prematurely discarded.

For a grid-based environment, several common heuristics are:

- Manhattan Distance: Calculates the distance by summing movements along grid axes. This heuristic is optimal in an environment where only cardinal movements are allowed.
- Chebyshev Distance: Measures the distance using the maximum number of moves along any axis. This heuristic is suitable in an environment where diagonal moves have the same cost as cardinal moves.
- Euclidean Distance: Calculate the straight-line distance between two points. This heuristic is admissible because no path between two points can be shorter than the straight line connecting them.

III. Algorithm Design

While this project utilizes the A* algorithm, which is standard and very well-known algorithm in pathfinding, its novelty and effectiveness do not stem from the choice of algorithm itself or the choice of heuristic. Rather, the improvement comes from the graph representation, specifically how nodes are generated.

A. *Euclidean Distance Heuristic*

A heuristic provides the informed part in a best-first search algorithm, like A*, guiding the algorithm efficiently toward the goal. In this project, the three-dimensional Euclidean distance was selected as the heuristic. The Euclidean distance is the ordinary straight-line distance between two points in space. For any two points $p_1 = (x_1, y_1, z_1)$ and $p_2 = (x_2, y_2, z_2)$, the Euclidean distance D is defined as:

$$D(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

This selection of this heuristic was justified by two of its properties, which are:

- Compatibility with 3D Free-Form Movement: Unlike the simpler two-dimensional grid worlds where movements are restricted to cardinal or diagonal steps, player movements in Minecraft are not rigidly locked into the voxel grid. As such, the Euclidean distance heuristic is the most accurate choice in such an environment because it measures the straight-line distance between two points irrespective of grid alignment.
- Guaranteed Admissibility: In order for an A* algorithm to find optimal solutions, the heuristic must be admissible. The Euclidean distance inherently satisfies this condition. A admissible heuristic must never overestimates the true cost. Since no possible path of movement between two points in a 3D space can be shorter than the direct line connecting them, this heuristic will always provide a cost estimate that is less than or equal to the actual cost.

B. *Physics-Aware Graph Representation*

The effectiveness of this algorithm comes not from the choice of A*, but from the specific way the game world is represented as a graph. A naive approach might define each empty block as a node and each adjacent empty block as its neighbors. The proposed design of this project abstracts the node, not as merely an empty block but a valid standing position. An edge between two nodes is not just a 1-block movement, but a complete, time-optimal action that transport the player from one standing position to another.

The ultimate goal of a speedrunner is to minimize time taken, not just distance travelled. The fastest mode of on-foot travel is known as sprint-jumping with an average speed of 7.127 blocks/second, almost 1.3x faster than sprinting normally. Due to this, the player essentially almost always want to sprint-jump to their destination. With sprint jumping, the player can move up to 4 blocks in a single action.

However, sprint-jumping is not always the best choice. Due to the fixed airtime of a jump, a jump covering less than three blocks could actually prove slower than simply sprinting across the same distance. Sprint-jumping also is not an action that is always available. In spaces with low ceiling, players might not have enough room that is necessary to perform a sprint-jump.

For any given node/state, the algorithm generates potential neighbors/directly accessible state by simulating the game's physics for a range of possible actions:

1) Multi-Block Horizontal Jumps

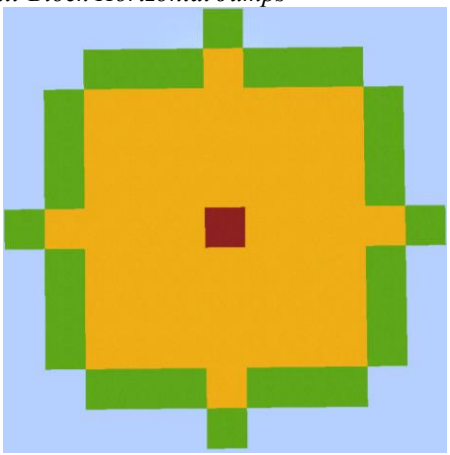

Fig. 3.1. Possible States on the Same Level
Source: Author

The blocks that are reachable from the block (represented by the red block) is shown in Figure 3.1. The algorithm projects the full trajectory of a sprint-jump, creating single edges which can span four blocks horizontally. This is represented by the green blocks. This directly models the most efficient way to cross open terrain. By jumping with less forward force, it is also possible to reach other blocks which is represented in Figure 3.1 by the yellow blocks.

2) Vertical Traversal

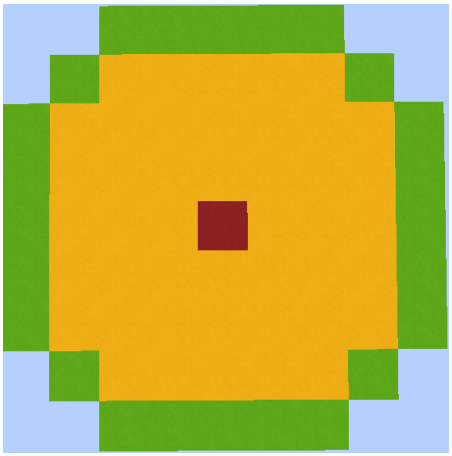

Fig. 3.2. Possible States on One Level Above
Source: Author

When jumping, a player can move up 1 block higher than where they currently are. Due to the parabolic nature of jumping, the amount of states reachable from the origin (red block) is less than the amount of states reachable if they were at the same level. In Figure 3.2, the green blocks represent the states that are reachable from one block below, assuming there are no obstacles in between. By lowering the forward force, the yellow blocks are also reachable. Although ascending more than one block is not possible under normal circumstances, descending a considerable distance is more than doable. Falling more than

three blocks will result in fall damage. This means that jumping while the targeted block is three blocks lower will results in fall damage, thus not ideal.

3) Trajectory Validation

A potential landing spot is not considered a valid neighbor unless the entire path to it is clear of obstructions. This is done by:

- Parabolic Arc Simulation: The algorithm does not check a simple straight line. It simulates the player's two-block-high hitbox moving along the parabolic arc of the jump. This accurately models how a player actually moves through the air.
- Intermediate Collision Checks: During this simulation, it queries intermediate coordinates along the arc to ensure that all blocks within the player's hitbox are non-solid (e.g., air). This check is what prevents the algorithm from suggesting a path that clips through a low-hanging tree leaf, the corner of a cliff, or a cave ceiling.

By building a graph where each edge represents an efficient, validated, real-world action, we transform the problem. Instead of asking A* to find the shortest path through a grid of millions of tiny steps, we ask it to find the optimal sequence of large, pre-calculated, high-speed movements. The Euclidean heuristic provides the global search strategy, but it is this intelligent, physics-aware node generation that provides the local, domain-specific expertise necessary for genuine optimization.

IV. System Implementation

A. Implementation Details

1) Programming Languange

The mod is written in Java. Java was chosen as it was the native language that Minecraft: Java Edition was written in. This choice provides a direct, performant access to the game's classes and method without the need of intermediate compatibility layers. This is essential for our algorithm, which must perform thousands of real-time physics calculations and world-state queries per second without impacting game performance.

2) Modding Toolchain

The Fabric modding toolchain for Minecraft has been growing swiftly in the last couple of years. This toolchain was chosen for its lightweight, modular nature compared to the more heavy Forge toolchain. Fabric provides a clean, stable API that is well-suited for complex modifications. The primary feature that it provides is Mixins, which are ways to inject bytecode without modifying preexisting properties. Its minimal performance overhead ensures that the pathfinding calculations, which can be computationally intensive, have the least possible impact on the game's overall frame rate and responsiveness.

B. User Workflows and Command Structure

Interaction with the pathfinding system is handled through a simple and intuitive command-based interface. This design choice allows for easy integration into the existing gameplay without requiring complex custom UI elements.

- Path Calculation (/pathto): The primary function is initiated via the /pathto <x> <y> <z> command. This command is registered using Fabric's CommandRegistrationCallback API. The command is designed to parse three integer arguments representing the target coordinates. Upon execution, the command retrieves the player's current position as the start node and the provided coordinates as the goal node, then triggers the A* pathfinding calculation on a separate thread to avoid freezing the game client.
- Path Visualization (/stoppath): Once a path is calculated, a client-side process begins rendering the path using particles. To provide the user with control over this visual effect, a second command, /stoppath, is implemented. This simple command terminates the particle visualization loop, clearing any existing visual guides from the screen.

C. Core Implementation

Below are the core implementations that are relevant to this paper:

1) Node Class

```
public class Node {
    private BlockPos pos;
    private Node parent;
    private double f;
    private double g;
    private double h;
    private boolean jump;
    private double jumpStrength;
    // Constructor
    public Node(BlockPos pos, Node parent, double f,
double g, double h, boolean jump, double jumpStrength)
{
        this.pos = pos;
        this.parent = parent;
        this.f = f;
        this.g = g;
        this.h = h;
        this.jump = jump;
        this.jumpStrength = jumpStrength;
    }

    public Node(BlockPos pos, Node parent, double g,
double h, boolean jump, double jumpStrength) {
        this.pos = pos;
        this.parent = parent;
        this.g = g;
        this.h = h;
        this.f = g + h;
        this.jump = jump;
        this.jumpStrength = jumpStrength;
    }

    // Getters
    ...
    // Setters
    ...
```

2) Node Generation

```
private List<Node> getNeighbors(Node currentNode,
BlockPos goal) {

    List<Node> neighbors = new ArrayList<>();

    // Walking moves
```

```
    addNeighbor(neighbors, currentNode, goal, new
BlockPos(1, 0, 0), false,
Constants.DEFAULT_WALKING_FORCE);

    ...

    // Jump-up moves

    // Short Jumps

    addNeighbor(neighbors, currentNode, goal, new
BlockPos(1, 1, 0), true, Constants.FORCE_SHORT_JUMP_B);

    ...

    // Short-Medium Jumps

    addNeighbor(neighbors, currentNode, goal, new
BlockPos(2, 1, 0), true, Constants.FORCE_SHORT_JUMP_A);

    ...

    // Medium Jumps

    addNeighbor(neighbors, currentNode, goal, new
BlockPos(2, 1, 2), true,
Constants.FORCE_MEDIUM_JUMP_B);

    ...

    // Long Jumps

    addNeighbor(neighbors, currentNode, goal, new
BlockPos(3, 1, 1), true, Constants.FORCE_LONG_JUMP_B);

    ...

    // Very Long Jump-Up

    addNeighbor(neighbors, currentNode, goal, new
BlockPos(4, 1, 2), true, Constants.FORCE_LONG_JUMP_A);

    ...

    // Falling moves

    addFallingNeighbors(neighbors, currentNode, goal);

    // Sprint-jump moves

    // Long-jump

    addNeighbor(neighbors, currentNode, goal, new
BlockPos(5, 0, 0), true, Constants.FORCE_LONG_JUMP_A);

    ...

    // Medium-jump

    addNeighbor(neighbors, currentNode, goal, new
BlockPos(4, 0, 0), true,
Constants.FORCE_MEDIUM_JUMP_A);

    ...

    // Short-jump

    addNeighbor(neighbors, currentNode, goal, new
BlockPos(3, 0, 0), true, Constants.FORCE_SHORT_JUMP_A);

         ...

    return neighbors;

}
```

3) A Search Algorithm*

```
public PathfindingResult findPath(BlockPos startPos,
BlockPos endPos) {

    PriorityQueue<Node> openSet = new
PriorityQueue<>(Comparator.comparing(Node::getF));

    Set<BlockPos> closedSet = new HashSet<>();

    Node startNode = new Node(startPos, null, 0,
MovementUtils.getEuclideanDistance(startPos, endPos),
false, 0);

    openSet.add(startNode);

    long startTime = System.currentTimeMillis();

    while (!openSet.isEmpty()) {

        if (System.currentTimeMillis() - startTime >
Constants.PATHFINDING_TIMEOUT_MS) {

                System.out.println("Pathfinding took
too long, timed out!");

                return new PathfindingResult(null,
closedSet.size());

        }

        Node currentNode = openSet.poll();

        if
(currentNode.getPos().isWithinDistance(endPos, 2.0)) {

                List<Node> path =
reconstructPath(currentNode);

                return new PathfindingResult(path,
closedSet.size());

        }

        closedSet.add(currentNode.getPos());

        for (Node neighborNode :
getNeighbors(currentNode, endPos)) {

                if
(closedSet.contains(neighborNode.getPos())) {

                        continue;

                }

                Node existingNode =
openSet.stream().filter(n ->
n.getPos().equals(neighborNode.getPos())).findFirst().o
rElse(null);

                if (existingNode == null ||
neighborNode.getG() < existingNode.getG()) {

                        if (existingNode != null) {
        openSet.remove(existingNode);

                        }

                        openSet.add(neighborNode);

                }

        }

    }

    return new PathfindingResult(null,
closedSet.size()); // No path found

}
```

V. Result and Analysis

A. Experimental Setup

To ensure a rigorous and reproducible evaluation, a suite of standardized test cases was designed. All tests were conducted

on Minecraft: Java Edition version 1.21.5, using the specific world seed 8558294790622361916 to control for environmental variables. The scenarios were chosen to test the algorithm's performance against varying levels of distance, terrain complexity, and environmental density.

B. Performance Metric

The performance of the pathfinding mod was evaluated using the following metrics:

- Path Length: The total number of nodes (blocks/waypoints) in the final, generated path.
- Nodes Expanded: The total number of nodes retrieved from the Open Set and evaluated by the algorithm. This is a direct measure of the size of the search space explored and indicates the algorithm's computational effort.
- Calculation Time (ms): The wall-clock time from the moment the pathfinding request is initiated to the moment the complete path is returned.

C. Results

After conducting the test and compiling the results, below the results from the five test scenarios are presented in Table 5.1 as such:

Test	Dimension	Scenario	Node Count	Node Expanded	Time (ms)
1.	Overworld	Hilly Terrain	25	355	5753
2.	Overworld	Flat Terrain	18	216	9138
3.	Nether	Dangerous Terrain	19	127	1781
4.	Nether	Fortress Navigation	22	157	2642
5.	Overworld	Stronghold Navigation	31	418	3993

Fig. 5.1. Test Results
Source: Author

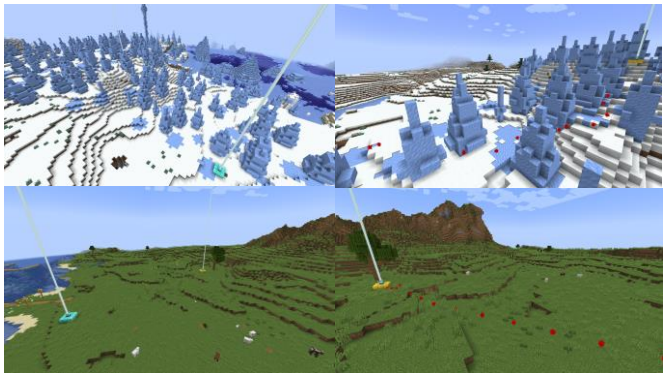

Fig. 5.2. Test Images
Source: Author

D. Analysis

The collected data provides several key insights into the algorithm's performance characteristics. The most intuitive correlation is between environmental complexity and computational effort. Test 1 (Overworld Hilly) and Test 5 (Overworld Stronghold), which feature significant verticality and cluttered, enclosed spaces respectively, required the algorithm to expand a large number of nodes (355 and 418) relative to the final path length. This is expected behavior, as complex terrain forces the A* search to explore many suboptimal branches before converging on the best route.

In contrast, the Nether-based scenarios (Test 3 and Test 4) were computationally very efficient, with the lowest number of expanded nodes and the fastest calculation times. This is likely due to the nature of Nether terrain, which, while dangerous, often features large open caverns and fewer small, fiddly obstacles than a forested Overworld biome. The pathfinding in these relatively open spaces is more direct, allowing the heuristic to guide the search more effectively.

A notable anomaly is Test 2 (Overworld Flat). Despite having the simplest terrain, the lowest path length, and a low number of expanded nodes, it recorded the longest calculation time (9138 ms). This suggests that factors outside of the core algorithm's complexity, such as the overhead of loading new world chunks over a longer distance, can have a significant impact on real-world performance. The path may be simple to calculate, but if the necessary world data is not readily available in memory, the process will be bottlenecked by I/O operations.

Overall, the results confirm that the physics-aware node generation is effective at finding short, coherent paths through complex 3D environments. However, the performance is clearly dependent on both the local complexity of the search space and larger game engine factors like chunk loading.

VI. Conclusion

The research successfully demonstrates that the A* search algorithm, when coupled with a highly specialized, physics-aware graph representation, can be applied within the game's complex, procedurally generated 3D environment to compute and visualize efficient paths in near real-time. The development of a functional Fabric mod serves as a concrete proof-of-concept, and its performance was quantitatively analyzed across a range of scenarios, confirming its viability as a tool for route optimization. There are a couple of important things that can be noted:

A. Limitations

Despite its successes, this work has several key limitations that must be acknowledged. These limitations stem from the simplifications made to render the problem computationally tractable and provide a clear baseline for analysis.

- Heuristic Simplification: The core limitation is the use of a purely geometric Euclidean distance heuristic. As demonstrated in the analysis, the shortest path is not always the fastest path. The current system does not account for variable traversal costs associated with different terrain types, the significant time penalty of mining through obstacles, or the ability of a player to place blocks to create bridges and shortcuts. The resulting paths are optimal in distance, but may be suboptimal in time.
- Static Pathing: The algorithm generates a single, static path based on the world state at the moment of calculation. Minecraft is a highly dynamic environment where obstacles like mobs are in constant motion and the player continuously alters the terrain. A path that is valid at one moment can become blocked or suboptimal in the next. The lack of a dynamic re-planning mechanism limits the tool's utility in a live, interactive gameplay session.
- Abstract Path Representation: Finally, the tool's output is purely representational, not prescriptive. It displays a visual line indicating the optimal sequence of locations but does not translate this route into the specific, timed sequence of player inputs required to follow it. For instance, while the path may incorporate a complex sprint-jump between two distant nodes, the visualizer simply connects these points; it offers no guidance on the required run-up, the precise timing of the jump, or the necessary mouse movement. This leaves the complex mechanical execution entirely to the player's own skill and knowledge, creating a potential gap between the theoretically optimal path and a player's ability to implement it.

B. Potential For Improvements

The limitations identified above point directly to several promising avenues for future research. These extensions would build upon the foundation established in this paper to create a more powerful and practical speedrunning assistance tool.

1. Implementing Dynamic Re-planning: To address the issue of static pathing, a re-planning module should be integrated. This could take several forms, such as "path splicing," where the algorithm periodically re-calculates the next few steps of the path to navigate around immediate, unforeseen obstacles. Another approach would be event-triggered re-planning, where the system listens for game events and automatically triggers a recalculation.
2. Goal-Oriented Pathfinding: The system could be expanded from simple point-to-point navigation to a more intelligent, goal-oriented planner. Instead of requiring the user to provide exact coordinates for an endpoint, a future version could accept high-level commands like "find nearest Bastion." This would require the algorithm to first perform a search for a valid goal state within a given radius before calculating the path to it.
3. Automated Path Traversal: To bridge the gap between path visualization and practical execution, a future development would be the creation of an automated traversal bot. This system would consume the generated path and translate each node into the precise sequence of inputs required to navigate it. By leveraging the movement data already stored in each path node, the bot could perform complex maneuvers with a high degree of precision, effectively transforming the tool from a navigational guide into an autonomous agent.

Appendix

Full source code of the program is available at: https://github.com/DanielDPW/Makalah_IF2211_Stima
Video link of the demonstration is available at: https://youtu.be/RJbxJmaREh8

Acknowledgment

The author expresses gratitude to lecturer Dr. Rinaldi Munir, M. T. as the lecturer of the IF2211 Algorithm Strategies course, for his guidance and resources provided in finishing this paper. Special thanks are due to friends and fellow students of the Algorithm Strategies class. The insightful discussions, collaborative problem-solving and constant encouragement made the challenges of this project much more manageable.

The author would also like to acknowledge the wider community whose work made this research possible: Mojang Studios for developing Minecraft, a rich and complex environment for algorithmic exploration and the developers of the Fabric modding toolchain for creating powerful and accessible tools.

Finally, the author wishes to express their deepest appreciation to their family for their unconditional love, patience, and unwavering support throughout the entire process of writing this paper.

REFERENCES

[1] R. Munir. "Penentuan Rute Bagian 1: BFS, DFS, UCS, Greedy Best First Search", 2025. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf

[2] R. Munir. "Penentuan Rute Bagian 2: Algoritma A*", 2025. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf

[3] Fabric Documentation, https://docs.fabricmc.net/

[4] Minecraft Wiki, https://minecraft.wiki/

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 24 Juni 2025

Daniel Pedrosa Wu (13523099)