Analysis of A*-Based Mob Pathfinding Behavior in Minecraft

Nathan Adhika Santosa - 13524041
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: nathan.santosa37@gmail.com, 13524041@std.stei.itb.ac.id

Abstract*—This paper presents an analysis of A*-based mob pathfinding behavior in Minecraft Java Edition through in-game experimentation and Python simulation. Minecraft's pathfinding engine is examined through decompiled source code, revealing a Weighted A* implementation with a 1.5x heuristic inflation factor, a hard node expansion limit tied to the zombie's follow range, and periodic path recalculation intervals. Four terrain scenarios, that is open terrain, U-shape obstacle, graph-like map, and a maze with mixed terrain penalties, were tested in-game using a Zombie as the test subject and compared against Dijkstra, Breadth-First Search, and Greedy Best-First Search implemented in Python. Results show that A* consistently produces optimal-cost paths with fewer node expansions than uninformed methods, while the Weighted A* modification in Minecraft trades occasional path suboptimality for real-time computational efficiency. The findings validate A* as the most suitable pathfinding algorithm for voxel-based game environments and confirm that Minecraft's implementation represents a well-justified engineering trade-off.

Keywords—A* algorithm, pathfinding, Minecraft, mob AI, Weighted A*, Dijkstra, BFS, GBFS, game AI.

I. Introduction

Pathfinding is one of the fundamental challenges in artificial intelligence, particularly in the context of game development. The ability of non-player characters (NPCs) and autonomous agents to navigate a dynamic environment efficiently and intelligently determines much of the perceived realism and gameplay quality of a game. Among the many pathfinding algorithms that have been developed, the A* (A-star) algorithm stands out as one of the most widely adopted due to its balance between optimality and computational efficiency.

Minecraft, developed by Mojang Studios, presents a uniquely interesting environment for studying pathfinding behavior. Unlike traditional game worlds, Minecraft's world is procedurally generated, three-dimensional, and discretized into a uniform grid of 1×1×1 meter blocks. This block-based structure imposes specific constraints on movement — entities must navigate around solid blocks, traverse varying terrain elevations, and in some cases cross water or avoid hazards such as lava. These characteristics make Minecraft's world a well-defined graph problem, where each block occupies a node and adjacency represents traversability between nodes.

Minecraft populates its world with a variety of *mobs* — the term used for mobile entities including passive creatures, hostile monsters, and neutral animals. Each mob type exhibits distinct navigation behavior; a zombie, for instance, pursues the player through complex terrain, while a spider can climb vertical surfaces.

This paper presents an analysis of A*-based pathfinding as applied to mob navigation in Minecraft. We examine the theoretical foundations of the A* algorithm, discuss its suitability for grid-based environments, and compare its behavior against alternative pathfinding strategies such as the Dijkstra Algorithm, Greedy Best First Search, and Breadth First Search. To validate our analysis, we conduct pathfinding experiments within Minecraft, measuring path quality, node expansion, and computational cost across different terrain configurations and mob scenarios.

II. Theoretical Foundation

A. Pathfinding in Games

Pathfinding has long been a central concern in game artificial intelligence, serving as the backbone of NPC behavior and autonomous agent navigation. Early game engines relied on simple grid-based movement with uninformed search strategies; however, as game environments grew in complexity, more sophisticated algorithms became necessary. The widespread adoption of A* in game development was largely driven by its introduction by Hart, Nilsson, and Raphael [1], who demonstrated that combining a cost function with an admissible heuristic yields optimal paths with significantly reduced node expansions compared to uninformed methods.

Unlike other games, which use navigation mesh systems, Minecraft is more naturally modeled as a three-dimensional grid graph, where each block position represents a node and edges represent valid movement transitions between adjacent positions.

B. Minecraft & Mobs

Minecraft is a sandbox video game developed by Mojang Studios, first released in 2011, in which players explore and interact with a procedurally generated three-dimensional

world entirely composed of discrete cubic blocks, each measuring 1×1×1 meters in-game [3]. The world spans diverse biomes, ranging from forests and deserts to oceans and mountains. This game allows players to gather resources, craft tools, and construct structures with no fixed objectives. A defining characteristic of Minecraft's environment is its voxel-based spatial structure, where every terrain feature and obstacle is represented as a uniform block on a three-dimensional grid.

Minecraft populates its world with a variety of mobile entities, collectively referred to as *mobs*. Mobs, such as, Zombies, Skeletons, Creepers, and Spiders actively pursue players upon detection. There are also mobs who do not pursue players, but other mobs, such as foxes who hunt down chickens. Each mob type exhibits distinct locomotion capabilities: ground-based mobs navigate across solid terrain, aquatic mobs such as Squids and Guardians move freely through water, and flying mobs such as Phantoms and Bats operate in three-dimensional airspace without ground constraints [8]. For the purposes of this paper, the analysis focuses primarily on ground-based hostile mobs, whose navigation behavior is most directly analogous to classical grid-based pathfinding.

C. Graph Theory & Grid Based Pathfinding

Pathfinding problems are formally grounded in graph theory, where an environment is represented as a graph *G = (V, E)* consisting of a set of nodes *V* and edges *E* connecting adjacent nodes [4]. In the context of Minecraft, each traversable block position in the world corresponds to a node, and edges represent valid movement transitions between neighboring positions. Edge weights can be assigned to reflect movement costs, such as increased cost for traversing water or sloped terrain. This weighted graph formulation provides a common theoretical basis for comparing their performance in terms of path optimality and computational efficiency.

D. A Algorithm*

The A* algorithm is an informed search algorithm that finds the shortest path from a source node to a goal node in a weighted graph. A* evaluates each candidate node using the evaluation function:

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

where *g(n)* is the exact cost from the start node to node *n*, and *h(n)* is an admissible heuristic estimating the cost from *n* to the goal. Walking on a flat standard surface has a cost of one and walking diagonally costs about 1.414. The table below contains some of the addition traversal cost g(n) when walking through[9] :

Block	Additional Cost
Elevation Change	0.5
Water Block	7
Fire Block/Magma Block	15
Lava	∞
Close to Fire	7

Table 2.4 Additional Cost for Traversing Different Condition

A heuristic is admissible if it never overestimates the true cost, which guarantees that A* returns an optimal path.

$$h(n) \leq h^*(n)$$

h(n) being the heuristic value and h*(n) being the actual cost. Common heuristic choices for grid environments include the Manhattan distance, Euclidean distance, and Chebyshev distance, each suited to different movement models. In the context of minecraft, ground mobs generally use the Euclidian distance. The Euclidian distance used in Minecraft differs from the general formula, where Minecraft includes the third dimensions towards the equation. [6]

$$h(n) = \sqrt{((x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2)}$$

The variables x, y, and z correspond towards the first, second, and third dimension in Minecraft.

E. Djikstra's Algorithm

Dijkstra's algorithm [2] is an uninformed shortest-path algorithm that expands nodes in order of their cumulative cost *g(n)* from the source, without any heuristic guidance. It guarantees optimal paths in graphs with non-negative edge weights and serves as a theoretical baseline for evaluating the efficiency gains provided by heuristic-guided methods such as A*. In the context of pathfinding, A* can be viewed as a generalization of Dijkstra's algorithm where the heuristic *h(n)* is set to zero.

F. Breadth First Search Algorithm

Breadth-First Search (BFS) is an uninformed graph traversal algorithm that explores nodes level by level, expanding all neighbors of the current node before moving to the next depth level. BFS uses a queue (FIFO) data structure to manage the order of node exploration. BFS guarantees finding the shortest path in terms of number of steps in an unweighted graph, but does not account for edge weights, making it unsuitable for weighted environments where terrain costs vary [7].

G. Greedy Best First Search

Greedy Best-First Search (GBFS) is an informed search algorithm that guides its search purely using a heuristic function h(n), selecting the node that appears closest to the goal at each step without considering the accumulated cost

g(n). While GBFS typically expands fewer nodes than A* or Dijkstra due to its aggressive goal-directed behavior, it does not guarantee an optimal path as it ignores the cost already incurred to reach the current node [5].

III. Research Methodology

A. Environment Setup

All experiments were conducted in Minecraft Java Edition version 1.21.5 [3] using the Fabric mod loader with Carpet mod[11] being installed. The Carpet mod was used primarily for its /tick freeze and /tick unfreeze commands, enabling precise control over when mob pathfinding begins in each test scenario. A flat superflat world was used as the base for all scenarios to eliminate terrain variability. Each scenario was constructed manually using concrete blocks of distinct colors: red concrete marks the goal position, black concrete marks the starting position, and white concrete marks the paths that could've been taken by the mob

B. Mob & Target Selection

A Zombie was selected as the test subject for all in-game experiments. This decision was made by the fact that all ground-based mobs in Minecraft share the same underlying A*-based pathfinding system [9], differing only in movement parameters. Using a single mob type therefore provides a consistent and representative sample without loss of generality. Both the Zombie and Villager were spawned using their respective spawn eggs in creative mode. The Villager was placed at the goal position (red concrete) and kept stationary by surrounding it with blocks to prevent movement, serving as a consistent and reusable target across all test runs.

C. Test Scenarios

Four distinct scenarios were designed to evaluate different aspects of A* pathfinding behavior, each built within the same flat world under identical conditions:

1. Open Terrain — No obstacles between start and goal. Serves as a baseline to verify that A* produces a straight optimal path under ideal conditions.
2. U-Shape — A U-shaped wall forces the mob to navigate around an obstacle that initially appears to block the direct route, testing heuristic behavior under misleading spatial conditions.
3. Graph-like Map — A map structured as a classical graph commonly used in algorithm textbooks and academic literature. This graph is similar to the one in the problem set for the Algorithm Strategy final exams.
4. Actual Maze with Mixed Penalties — A complex maze containing water, lava, fire, and varying elevation changes. Tests A*'s robustness in high-complexity environments with multiple types of movement penalties simultaneously.

D. Experimental Procedure

Each scenario was tested using the following procedure:

1. Initiate command /tick freeze
2. Place the zombie and villager in the black and red concrete.
3. Get in position to capture the zombie navigating to the villager using OBS.
4. Initiate command /tick unfreeze
5. Wait until the zombie reaches the villager to stop recording.

E. Comparison with Alternative Algorithms

For additional in-game algorithm observations, I have implemented a program in Python using grid representations that mirror each of the four in-game scenarios, which is used to compare other algorithms, such as Dijkstra, Greedy Best First Search, and Breadth First Search. In the program, the following metrics were recorded:

- Nodes expanded — total number of nodes visited before reaching the goal
- Path cost — total weighted cost of the final path
- Time to Target — time taken to compute the path in milliseconds and the amount of time it takes to get to the target

Time to Target is calculated by summing all step time in different terrains, in which step time has the formula

$$\text{Step\_time} = \text{distance}/\text{effective\_speed}$$

Where,

$$\text{Effective\_speed} = \text{zombie\_based\_speed} \times \text{Terrain\_Speed\_Multiplier[terrain\_B]}$$

The time to target during experimenting in Minecraft and time to target calculated in Python can differ because of the different methods used to get the results. The results during experimenting are manually taken by reviewing the recording, which almost always indicates variety in the results.

This python program is used to compare other algorithms other than A* because pathfinding algorithms in Minecraft mobs cannot be easily changed and the usage of mods is needed. I have tried to find mods in the community of Minecraft that implements different pathfinding algorithms for mobs, but I have not found any mods that does so. Since the time to finish this paper is limited, I have opted to use a python program for my comparisons.

IV. Results & Discussion

All image results from the python program will be in a google drive folder in the appendix section, because of the limited space given to make this paper. Image results shown in this paper are results that differ during experimenting in Minecraft.

Analysis towards Minecraft's source code is done by downloading from a github repository [10]. The reason behind this is that Minecraft doesn't release its original source code, so the usage of github repositories from the Minecraft modding community is necessary. The repository was then opened using IntelliJ IDEA to inspect the source code within the external libraries.

A. Results per Scenario

I. Open Terrain Scenario

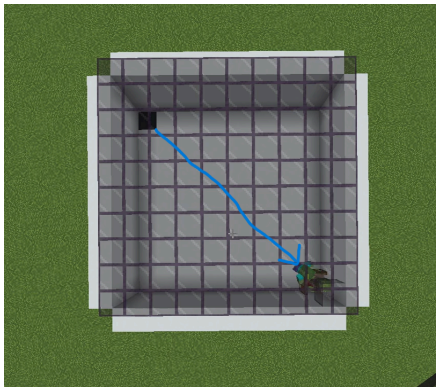


Figure 4.1 Open Terrain Scenario

In this scenario, the Zombie went straight to the villager in a straight line, diagonally from its spawn location to the villager. The total time for the Zombie to calculate the path move towards the villager is 6.217 seconds.

Using the python program to compare different algorithms towards the A* algorithm gives us this result

Algorithm	Nodes	Cost	Path Length	Time to Target(ms)
A*	10	12.7260	10	6363.3763
Dijkstra	100	12.7260	10	6365.6741
BFS	100	12.7260	10	6366.0344
GBFS	10	12.7260	10	6363.5459

Table 4.1 Program Results from the Open Terrain Scenario

From the results, A* and GBFS has the lowest Time to Target and the smallest amount of nodes explored. However, A* has the lowest Time to Target, so it can be concluded that A* is the best algorithm for an open terrain scenario.

II. U-Shape Scenario

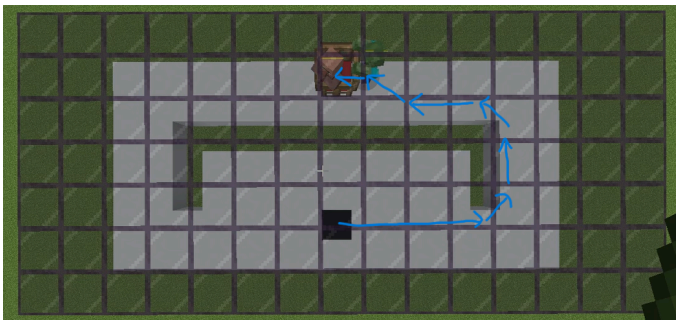


Figure 4.2 U-Shape Scenario

In this scenario, the zombie looped around the gap in the obstacle to reach the villager. The total time for the zombie to reach the villager, including pathfinding, is 7.166 seconds.

Using the python program to test A* algorithm against other algorithms on the U-Shape Scenario, we get these results

Algorithm	Nodes	Cost	Path Length	Time to Target
A*	81	15.2420	15	7635.8811
Dijkstra	90	15.2420	15	7623.8252
BFS	90	15.2420	15	7622.6854
GBFS	47	15.2420	16	8744.3540

Table 4.2 Program Results from the U-Shape Scenario

From these results, BFS has the lowest time to target compared to the other algorithms but has the most number of nodes explored because of its breadth characteristics. GBFS has the lowest number of nodes explored, but the drawback is that the time to target and path length is the largest. GBFS only uses h(n) for its f(n), therefore this algorithm moves the zombie forward(h(n) is lower moving forward before looping the gap, adding the number of blocks and time traveled.

A* has more balanced results. A* does not have the lowest or the highest number of nodes explored and time to target.

III. Graph like Map

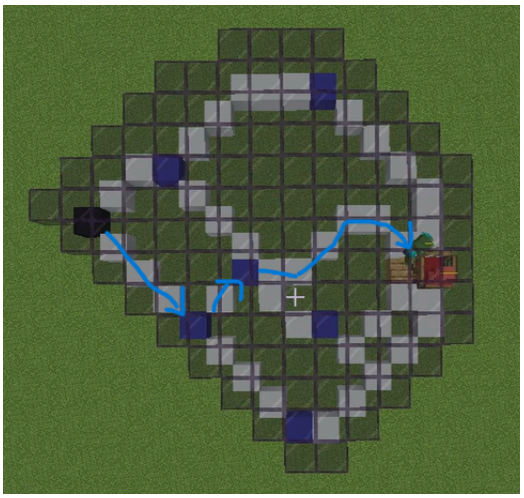


Figure 4.3.1 Graph like Map Scenario

In this scenario, the zombie goes to the node in its southeast direction(marked with the blue concrete). The zombie later went to the node in the middle and then straight to the zombie. The zombie reached the villager in 7.93 seconds.

From the python program, we get these results

Algorithm	Nodes	Cost	Path Length	Time to Target
A*	27	17.1400	14	8579.9271
Dijkstra	47	17.1400	14	8570.4530
BFS	47	17.1400	14	8570.3986
GBFS	14	17.1400	14	8570.1688

Table 4.3 Program Results from the Graph-Like Scenario

In this scenario, we can conclude that A* has the worst time to target out of the three other algorithms, even though A* doesn't have the most amount of nodes expanded.

In the program, the A* algorithm choose a different path than the one that the zombie in minecraft took. Here is the path that the A* in the program took

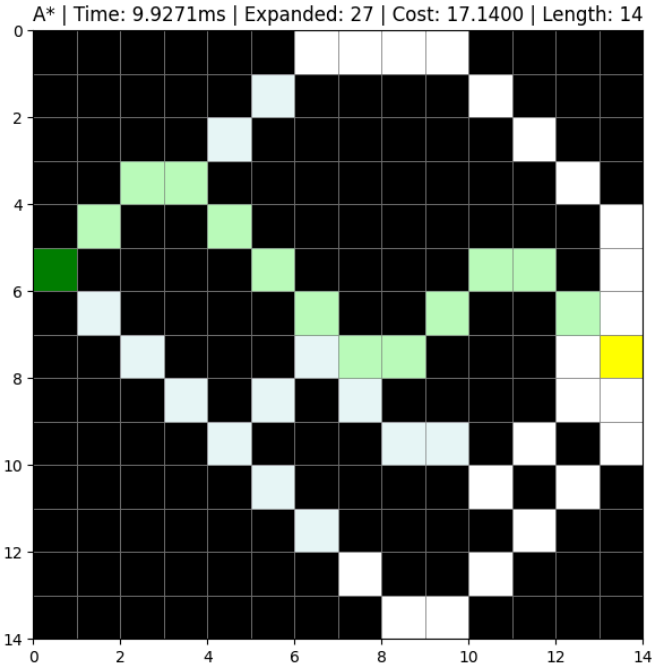


Figure 4.3.1 A* Result from Python Program

The A* algorithm went northeast before going southeast and then went straight to the target. However, it turns out that the number of nodes used to reach the goal is the same and with the same amount of cost, therefore the path taken here and during experimenting is a matter of choice. This might be the reason behind A*'s longer time to target. The algorithm had to calculate both paths that are both the optimal route and with the algorithm having a more costly heap operation, therefore A* had the longest time to target.

IV. Maze with Mixed Penalties

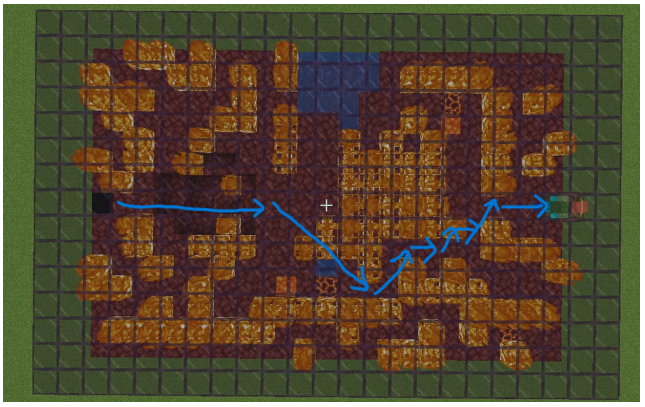


Figure 4.4.1 Maze with Mixed Penalties Scenario

The zombie, in this scenario, goes forward going through the ditch. After the zombie is half way to the goal, the zombie went through some fires and clear paths sustaining damage before reaching its goal. The zombie chooses this path despite there being paths without sustaining or at least minimizing damage, like the north or south path. The zombie reach its target in 13.767 seconds.

Algorithm	Nodes	Cost	Path Length	Time to Target
A*	170	36.3820	32	18197.6250
Dijkstra	289	36.3820	32	18197.1182
BFS	331	104.0000	23	14009.5274
GBFS	23	104.0000	23	14000.5130

Table 4.4 Program Results from the Maze Scenario

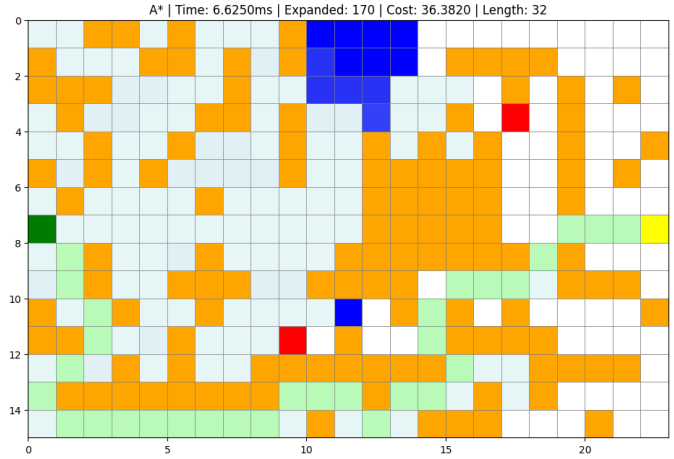


Figure 4.4.2 A* Results for the Maze map

From the results, it can be concluded that A* and Dijkstra had the lowest cost compared to BFS and GBFS, and Dijkstra has the least time to target compared to A*. Despite going through more nodes than A*, it still had a lower time to target than A*. A* and Dijkstra used the same path, while GBFS and BFS also used the same path, that is a straight line from start which means the zombie went through multiple fire blocks (netherrack with fire on top of it) therefore increasing the amount of cost to reach the target.

The A* algorithm both in this experiment and the result from the python program always results in sustaining damage from the fire. In the program, it doesn't consider the hitboxes in Minecraft, so the zombie will sustain damage walking in a diagonal direction, like the one in the bottom left. While hitboxes in Minecraft is considered, Mobs in Minecraft has a block limit of 35 blocks for the A* algorithm(Depth Limited A*) to calculate the correct path. That is why during experimenting, the zombie just went through the fire even though there is a safe path to the villager.

B. Discussion

While empirical observations highlight how the agent's behavior deviates from pure A* logic, they do not explain the mechanics behind it. Therefore, the following discussion dissects the decompiled source code of Minecraft's pathfinding engine to understand its internal architecture.

The source code analysis of Minecraft's pathfinding system reveals that its implementation is fundamentally rooted in the A* algorithm. The core structure in PathFinder.java follows the canonical A* evaluation function f(n) = g(n) + h(n), where g(n) represents the accumulated cost from the start node and h(n) is the heuristic estimate to the goal. The Node.java class explicitly stores these values as separate fields g, h, and f, and the parent pointer is maintained through Node cameFrom directly mirroring the theoretical formulation of A*. The neighbor expansion loop in PathFinder.java updates g(n) only when a better path is found, which is the signature feature of A*'s optimality condition:

```
float tentativeGScore = current.g + distance +
neighbor.costMalus;
if (neighbor.walkedDistance < maxPathLength &&
  (!neighbor.inOpenSet() || tentativeGScore < neighbor.g))
{
  neighbor.cameFrom = current;
  neighbor.g = tentativeGScore;
  ...
}
```

The heuristic function getBestH() uses three dimensional Euclidean distance via from.distanceTo(target), which computes $\sqrt{(dx^2 + dy^2 + dz^2)}$ as confirmed in Node.java. This is consistent with the theoretical requirement that the heuristic must reflect the true geometry of the search space. In Minecraft's three-dimensional block grid, Euclidean distance is the natural lower bound for movement cost.

However, a critical deviation from standard A* is found in how the heuristic is applied. Rather than using the raw Euclidean distance, Minecraft inflates it by a factor of 1.5:

```
neighbor.h = this.getBestH(neighbor, targets) * 1.5F;
```

This renders the heuristic inadmissible, since h(n) may now be exceed h*(n), the true cost to the goal. In A* theory, admissibility is the necessary condition for guaranteeing an optimal path. By violating this condition, Minecraft's pathfinding implementation becomes what is known as Weighted A* (WA*), a variant that trades optimality for computational speed. In a game environment running at 20 ticks per second where mob responsiveness is critical, this trade-off is somewhat justified.

The cost function implementation further confirms the A* foundation. Terrain penalties are stored as costMalus in each Node and applied directly to the g(n) accumulation, consistent with the weighted graph formulation that A* requires. The penalty values(added with the cost value traversing) extracted from WalkNodeEvaluator.java and corroborated by the official Minecraft Wiki, water = 8, danger-fire (adjacent to fire) = 8, damage-fire (standing on fire) = 16, and lava = -1 (impassable), closely match the cost function implemented in the simulation, where water = 8, adjacency lava penalty = 8, fire = 16, and lava = 32. The only meaningful divergence is on lava, which is treated as fully impassable in Minecraft but assigned a high finite cost in the simulation as a simplification.

This consistency validates that the simulation's cost function is a somewhat approximation of Minecraft's pathfinding behavior.

The spatial awareness system in WalkNodeEvaluator.java validates movement in a diagonal direction. Diagonal movement is not freely permitted, it undergoes a two-stage validation. First, both orthogonal neighbors in the diagonal's component directions must be evaluated. Second, isDiagonalValid() applies hitbox-aware corner-cutting rules:

```
if (!(this.mob.getBbWidth() > 1.0F) ||
   !(ew.costMalus > 0.0F) && !(ns.costMalus > 0.0F)) {
   return (ns.y < pos.y || ns.costMalus >= 0.0F ||
canPassBetweenPosts)
      && (ew.y < pos.y || ew.costMalus >= 0.0F ||
canPassBetweenPosts);
} else {
   return false;
}
```

Large mobs with a bounding box width exceeding 1.0 block apply an OR condition, a diagonal is blocked if either orthogonal neighbor carries a positive penalty. Smaller mobs such as zombies (width 0.6 blocks) apply a more permissive rule. Minecraft additionally blocks diagonals when an orthogonal neighbor is elevated above the current position (ns.y > pos.y || ew.y > pos.y), this evaluation is absent from the simulation. The simulation permits diagonal movement unconditionally as long as the destination tile is not void. This difference explains the path divergence between Minecraft's zombie and the simulation, particularly in geometrically constrained scenarios such as the Maze map.

The most significant deviation from standard A* is the usage of a hard node expansion limit, identified through the combined analysis of Zombie.java and PathFinder.java. The zombie entity defines a FOLLOW_RANGE attribute of 35.0 blocks:

```
.add(Attributes.FOLLOW_RANGE, 35.0)
```

This value not only is used for the detection radius but also correlates with maxVisitedNodes, which imposes a hard ceiling on the number of nodes A* may expand before being forcibly terminated:

```
int maxVisitedNodesAdjusted =
(int)(this.maxVisitedNodes * maxVisitedNodesMultiplier);
while (!this.openSet.isEmpty()) {
   if (++count >= maxVisitedNodesAdjusted) {
      break;
   }
```

When this limit is reached before the goal is found, Minecraft does not return failure, instead it reconstructs the best partial path toward the nearest reachable node to the goal. This behavior directly explains the observed phenomenon in the Maze scenario, where the safe path avoiding fire requires a greater number of node expansions to discover the longer detour. When the node limit is reached first, Minecraft's A* selects the shorter suboptimal path that traverses fire blocks. The simulation, having no such constraint, consistently finds the globally optimal path. Furthermore, MeleeAttackGoal.java reveals that path recalculation does not occur every tick but rather every 4 to 10 ticks:

```
this.ticksUntilNextPathRecalculation = 4 +
   this.mob.getRandom().nextInt(7);
```

At 20 ticks per second, this corresponds to a recalculation interval of 0.2 to 0.5 seconds. This explains the occasional lag observed when a zombie appears slow to react to sudden target movement, the mob continues executing its previously computed path until the next recalculation occurs. The simulation does not model this behavior, as the path is computed once against a stationary target, representing an idealized condition without recalculation delay.

Having explained how Minecraft's A* implementation operates with its weighted heuristic, node expansion limit, hitbox-aware corner cutting, and periodic path recalculation. It becomes meaningful to evaluate how alternative pathfinding algorithms would perform under the same conditions.

Across all four scenarios, no single algorithm dominates in every metric, each exhibits distinct trade-offs between path optimality, node efficiency, and traversal time.

A* demonstrated the most balanced performance overall, consistently finding optimal paths with the lowest cost alongside Dijkstra while expanding significantly fewer nodes than BFS. In the Maze scenario, A* found the globally optimal path in simulation, yet the in-game zombie deviated by traversing fire blocks, directly explained by Minecraft's node expansion ceiling identified in the source code, where maxVisitedNodes forces a suboptimal partial path before the safer detour is discovered.

Dijkstra expanded more nodes than A* in every scenario but achieved lower Time to Target in the U-Shape and Maze scenarios, as its simpler g(n) only evaluation incurs less per-node overhead than A*'s combined f(n) computation when path lengths are similar. BFS expanded the most nodes and failed to account for terrain penalties, returning a path cost of 104.0000 versus A*'s 36.3820 in the Maze scenario, confirming its unsuitability for weighted environments. GBFS expanded the fewest nodes but not for cost in complex scenarios, because GBFS doesn't avoid penalties to reach its target. Confirming that its heuristic-only guidance trades path quality for speed,

The simulation's Time to Target values are consistently higher than in-game measurements due to fixed-speed assumptions

and manual timing variance, though the relative derivation between scenarios remains consistent, validating the simulation as a reasonable approximation.

Regarding the most optimal algorithm mob pathfinding, Minecraft's current Weighted A* implementation is well-justified for a real-time game environment. Among all tested algorithms, only A* and Dijkstra consistently produced optimal paths, but Dijkstra's uninformed expansion makes it computationally prohibitive for large or complex environments. GBFS is faster but sacrifices optimality. BFS is unsuitable for weighted terrain entirely. A* therefore remains the most appropriate choice and Minecraft's 1.5× heuristic inflation further optimizes it for real-time responsiveness at the cost of occasional suboptimality, a trade-off that is reasonable given the 20-ticks-per-second constraint of the game engine.

V. Conclusion

This paper analyzed the A*-based pathfinding behavior of mobs in Minecraft through both in-game experimentation and Python simulation across four terrain scenarios. The analysis confirmed that Minecraft's pathfinding engine is fundamentally rooted in A*, implemented as Weighted A* with a 1.5× heuristic inflation factor that trades path optimality for real-time responsiveness. Source code analysis further revealed three key deviations from standard A*. That is an inadmissible inflated heuristic, a hard node expansion ceiling tied to the zombie's FOLLOW_RANGE attribute, and periodic path recalculation every 4 to 10 ticks, all of which collectively explain the suboptimal in-game behavior observed in the Maze scenario.

Comparative evaluation against Dijkstra, BFS, and GBFS showed that no single algorithm dominates across all metrics. A* consistently produced optimal-cost paths with fewer node expansions than Dijkstra and BFS, while remaining more robust than GBFS in complex environments, environments with penalties. Among all tested algorithms, A* remains the most appropriate choice for Minecraft's pathfinding requirements, and its current Weighted A* implementation is well-justified given the computational constraints of a real-time game engine running at 20 ticks per second.

VI. Acknowledgement

The author would like to thank Dr. Rinaldi Munir as the lecturer of IF2211 Algorithm Strategy at Institut Teknologi Bandung, whose guidance and course material provided the theoretical foundation for this paper. The author also thanks his parents for their continued support and encouragement throughout the completion of this work.

VII. Appendix

- Minecraft Source Code : https://github.com/FabricMC/fabric-example-mod.git
- Experimentation Images & Videos & Results : https://drive.google.com/drive/folders/1YT8zvcbLYdsNFi335w-WDu1JjOzUL_4H?usp=sharing
- Python Algorithm Comparison Program: https://github.com/N8NAS/Analysis-of-A-Based-Mob-Pathfinding-Behavior-in-Minecraft.git

References

[1] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, Jul. 1968.

[2] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[3] Mojang Studios, *Minecraft Java Edition*, ver. 1.21.5, 2025. [Online]. Available: https://www.minecraft.net

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA: MIT Press, 2022.

[5] N. U. Maulidevi, "Penentuan Rute (Route/Path Planning) Bagian 1: BFS, DFS, UCS, Greedy Best First Search," Bahan Kuliah IF2211 Strategi Algoritma, Institut Teknologi Bandung, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/21-Route-Planning-(2026)-Bagian1.pdf

[6] N. U. Maulidevi and R. Munir, "Penentuan Rute (Route/Path Planning) Bagian 2: Algoritma A*," Bahan Kuliah IF2211 Strategi Algoritma, Institut Teknologi Bandung, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/22-Route-Planning-(2026)-Bagian2.pdf

[7] R. Munir and N. U. Maulidevi, "Breadth/Depth First Search (BFS/DFS) Bagian 1," Bahan Kuliah IF2211 Strategi Algoritma, Institut Teknologi Bandung, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/13-BFS-DFS-(2026)-Bagian1.pdf

[8] Minecraft Wiki Contributors, "Mob," *Minecraft Wiki*, 2025. [Online]. Available: https://minecraft.wiki/w/Mob

[9] Minecraft Wiki Contributors, "Mob AI," *Minecraft Wiki*, 2025. [Online]. Available: https://minecraft.wiki/w/Mob_AI

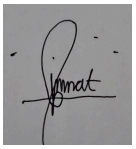
[10] FabricMC, "Fabric Example Mod," *GitHub*, 2025. [Online]. Available: https://github.com/FabricMC/fabric-example-mod

[11] gnembon et al., "Carpet Mod," *GitHub*, 2025. [Online]. Available: https://github.com/gnembon/fabric-carpet

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, 19 Juni 2026



Nathan Adhika Santosa - 13524041