The Applications of Pathfinding to Optimize Traversal in Minecraft

John Kim, Sean Paek, Akhilesh Bharatham

December 2023

Abstract

Minecraft's rise to video game stardom can be attributed to many factors, but one factor stands out: speedrunning. To speedrun Minecraft, players must walk across vast world generation and different biomes that feature different blocks. These blocks have the drastic potential to slow down the player's speed or cut them off from idealistic paths to get to their destination. This project explores path optimization by simulating pathfinding in Minecraft, considering the effect of various terrains and obstacles on player movement. We generated a random world, marked off a chunk of land, set arbitrary start and end points, and applied lattice paths, the pigeonhole principle, to determine the most efficient routes. Our findings suggest that while direct, straight-line paths are ideal in an open field, practical navigation must consider diagonal movements and circumnavigation of obstacles, with significant implications for speedrunning tactics and pathfinding algorithms.

Introduction

Minecraft's infinite terrain presents a labyrinth of challenges for players, particularly speedrunners aiming to complete objectives in the shortest possible time. Diverse biomes and block properties can either impede progress or offer shortcuts. Understanding the mechanics of Minecraft's world generation and pathfinding can substantially improve gameplay efficiency. Our study investigates the most effective pathfinding strategies, considering the game's block-based mechanics and environmental hindrances.

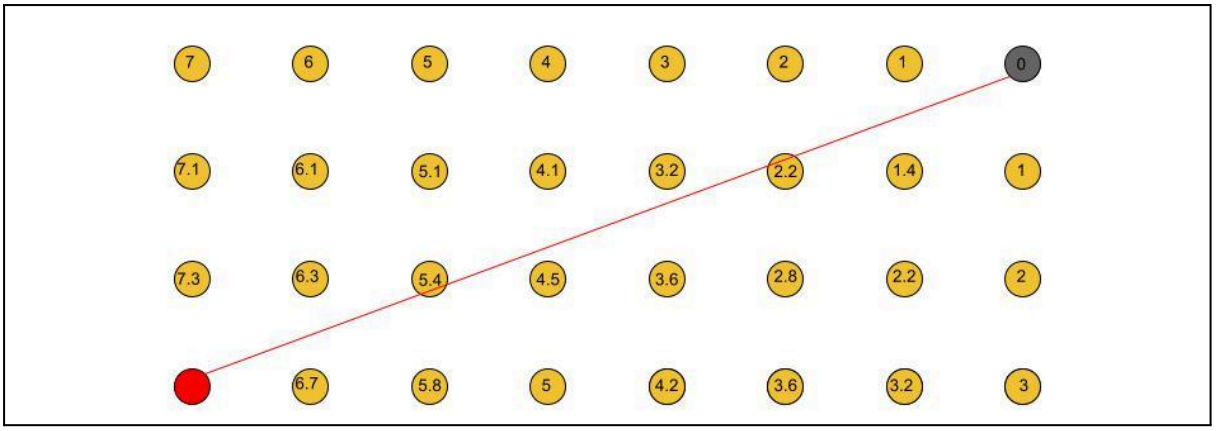
Analysis

The Problem

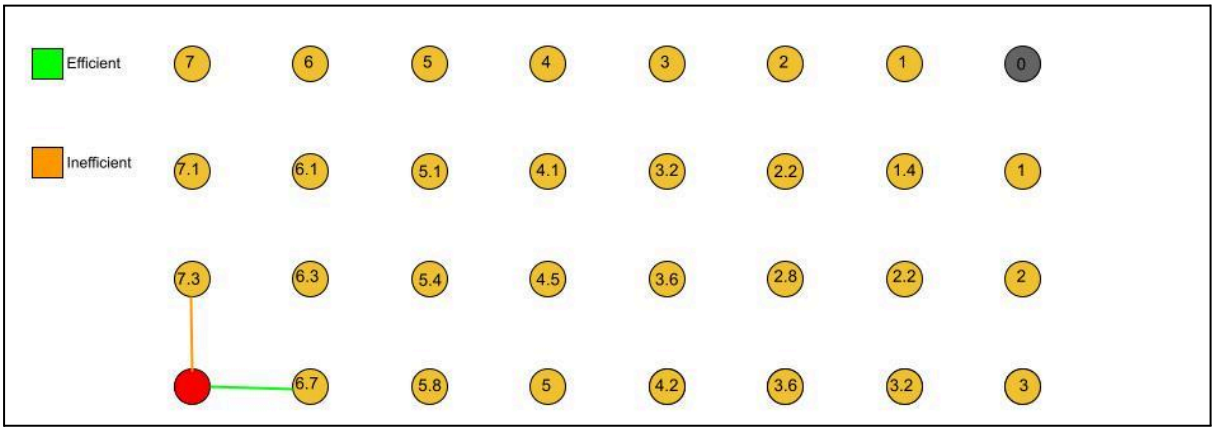
Let’s use a grid (with every dot representing a block) that represents the path from (0,0) to (8,4) to represent our problem. The red dot represents the origin, and the gray dot represents the destination.

Greedy First-Search Heuristic

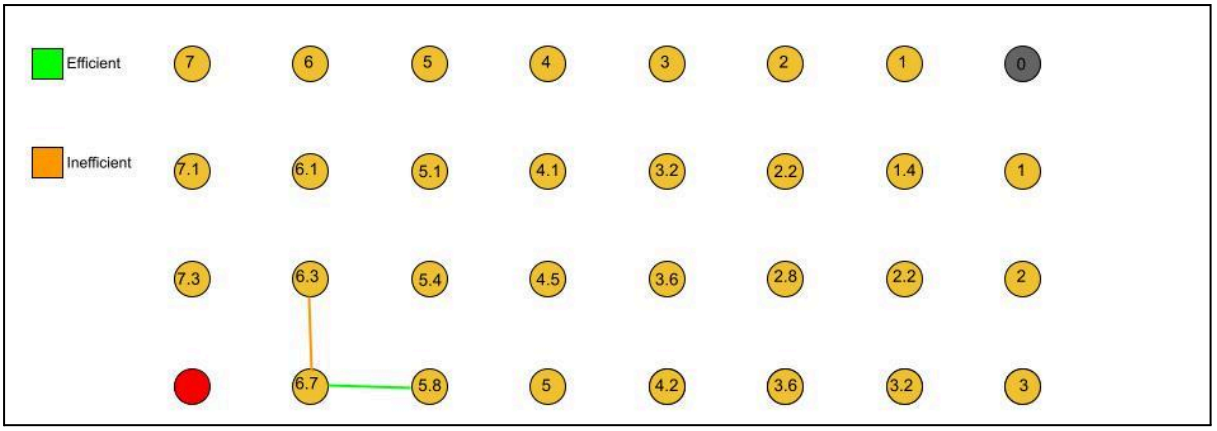
Let’s put a value on each block determined by the distance from the destination point.
The methodology of the Greedy First-Search Heuristic is to travel to blocks with the smallest distance from the end point. Since, the player can move diagonally in Minecraft, the shortest and most efficient route would be this:



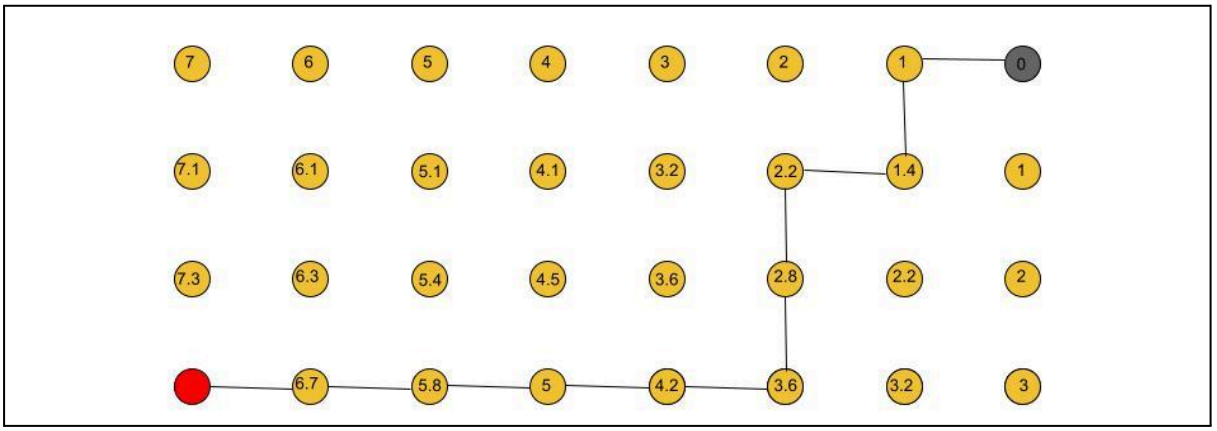
As you can see, the path from the starting point to the block with the smallest number is a straight path from the start to the destination point. However, this case is both trivial and obvious: the shortest and most efficient path from one point to another would be a straight line. Since we must later add constraints such as obstacles that either slow movement or completely inhibit movement, let us only consider movement that is solely orthogonal: north, south, east, and west.



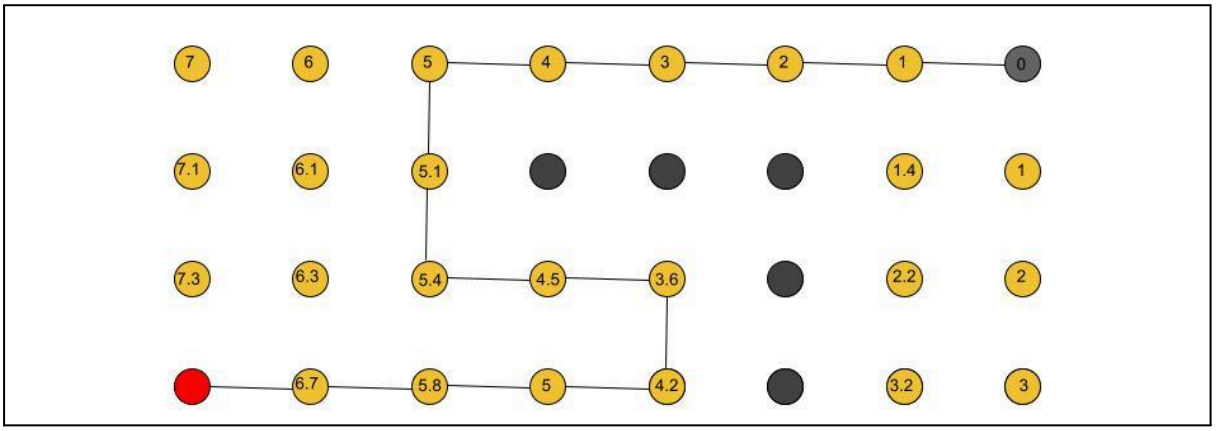
As we apply the Greedy First-Search Heuristic, we look to find the block with the smallest distance from the destination point and continue the pattern. From the start point, we have the choice to go to the block with a distance of 7.3 blocks and a distance of 6.7 blocks. Since we want to maximize movement, we want to go to the block with a shorter distance from the end point which would be 6.7. Now, we look to find the next block with the smallest distance from the destination point. From the “6.7 block” we have the choice to go to the “6.3 block” and the “5.8 block.”



Again, we choose the block with the smallest distance which is the “5.8 block.” We continue this heuristic until we reach the end to get the shortest path from the start to the end. Thus, we get the shortest route:



The main flaw comes when an obstacle impedes movement (represented by a black dot).

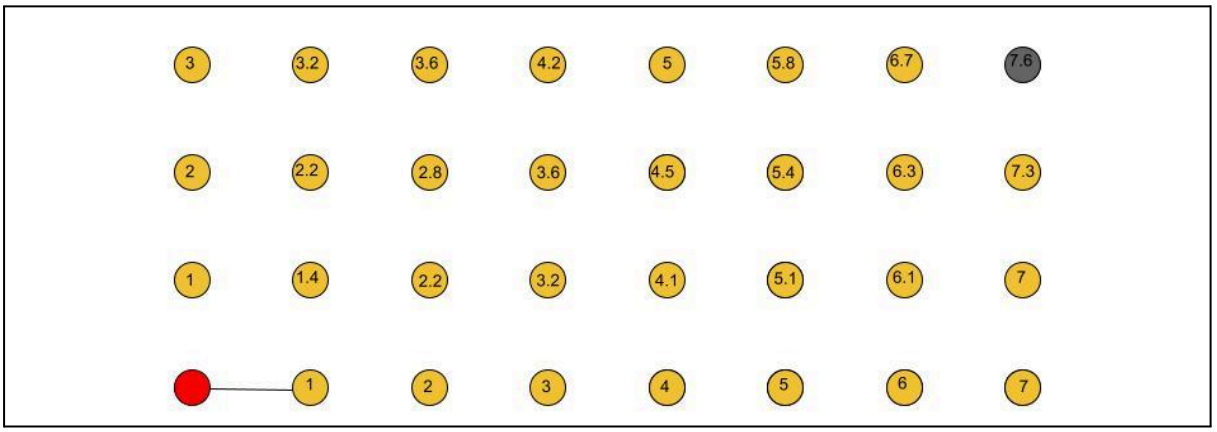


Simply by inspection, we can find a far more optimal route, but the heuristic only analyzes pathfinding one step ahead rather than seeing the problem as a whole.

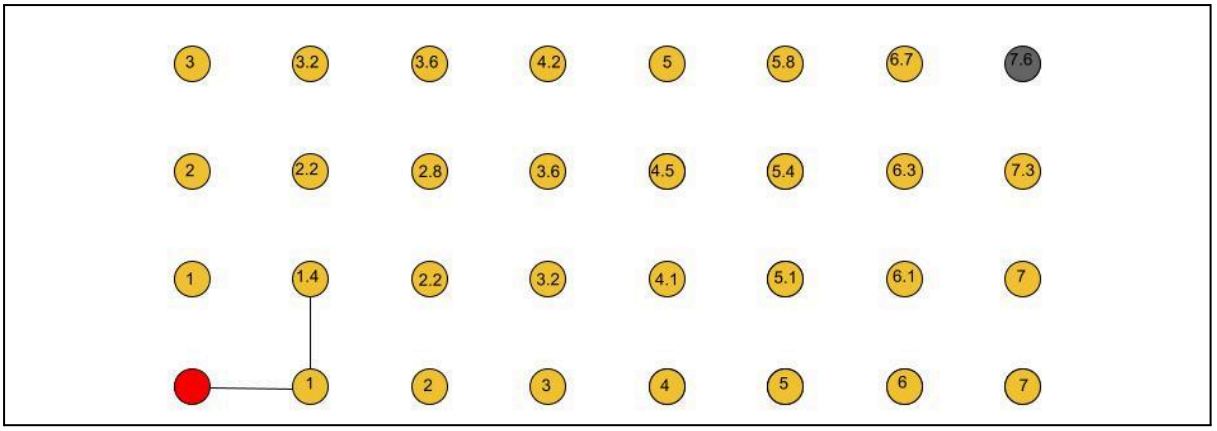
To account for pathfinding problems as a whole, let's try the Dijkstra's Algorithm approach.

Dijkstra's Algorithm

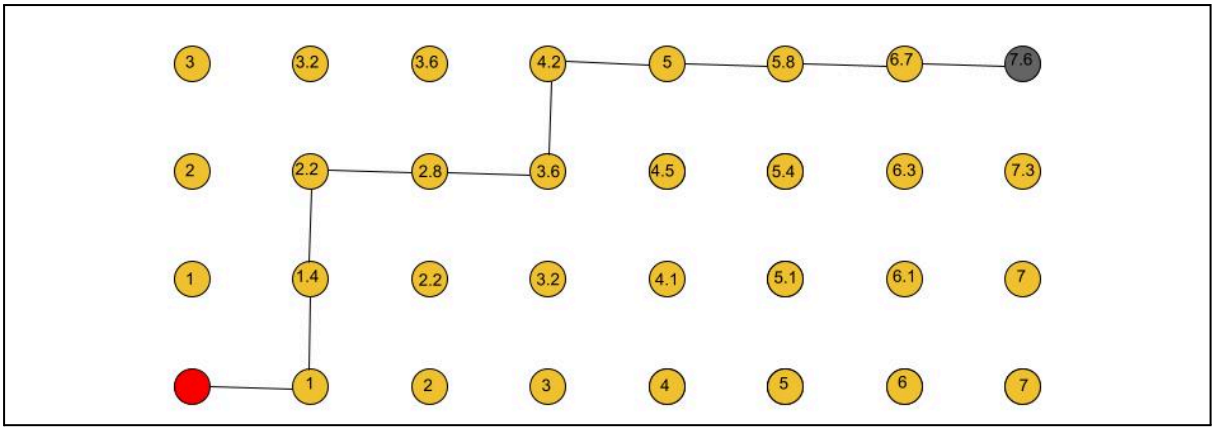
Dijkstra's algorithm scans every block we have not visited yet and prioritizes the blocks closest to the starting point as long as the blocks do not have a negative cost. So, we will display all the distances from the starting point to every block.



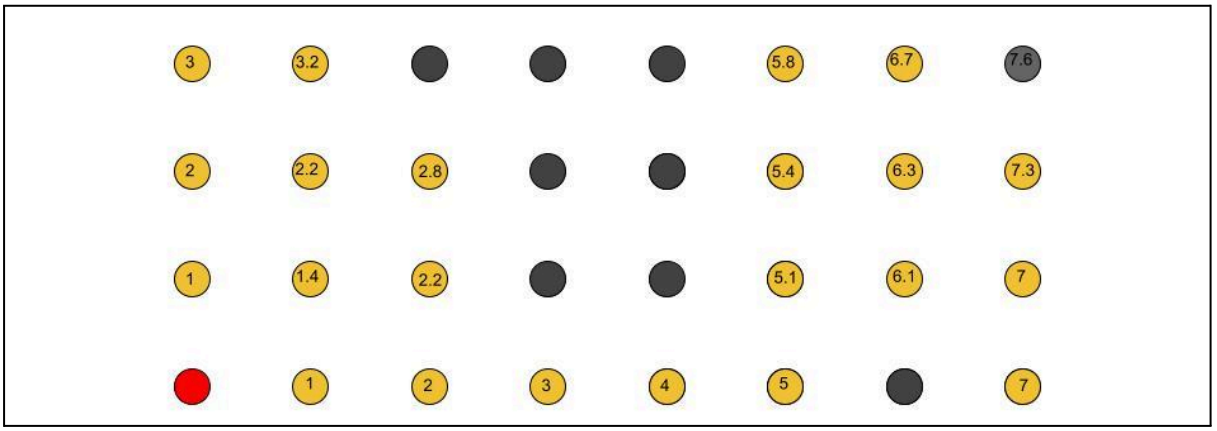
We first check the two nodes which have the shortest distance to the origin. Since both are 1 block away, we randomly choose one of the two. Now, we check the next unvisited blocks and find the next block with the shortest distance given that the number does not decrease.



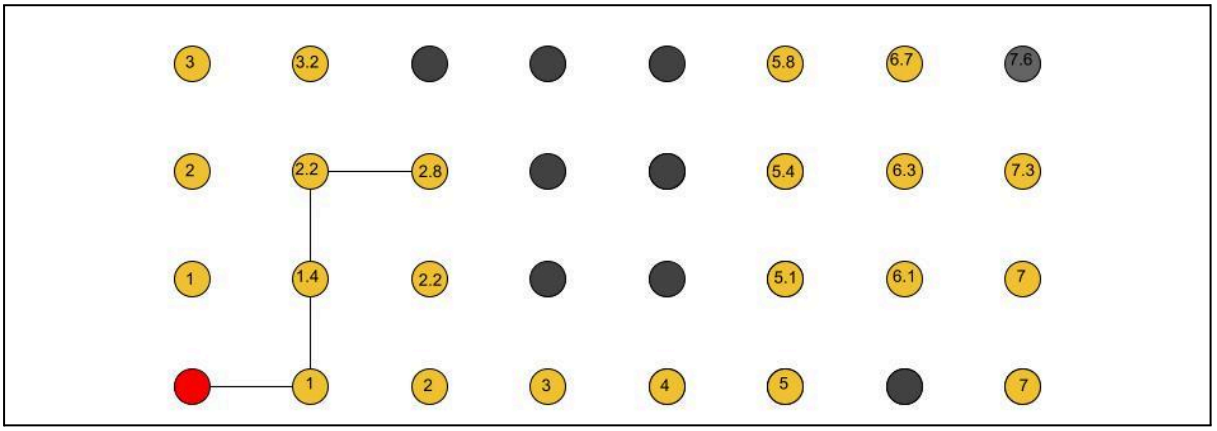
Since the "1.4 block" is closer than the "2 block" we travel to the "1.4 block." Now, we continue Dijkstra's algorithm in order to find the fastest path.



Using Dijkstra's algorithm, we are able to find the shortest path to the end point. The added benefit of Dijkstra's algorithm is that it can account for obstacles. Here is an example with black dots representing blocks that cannot be reached.

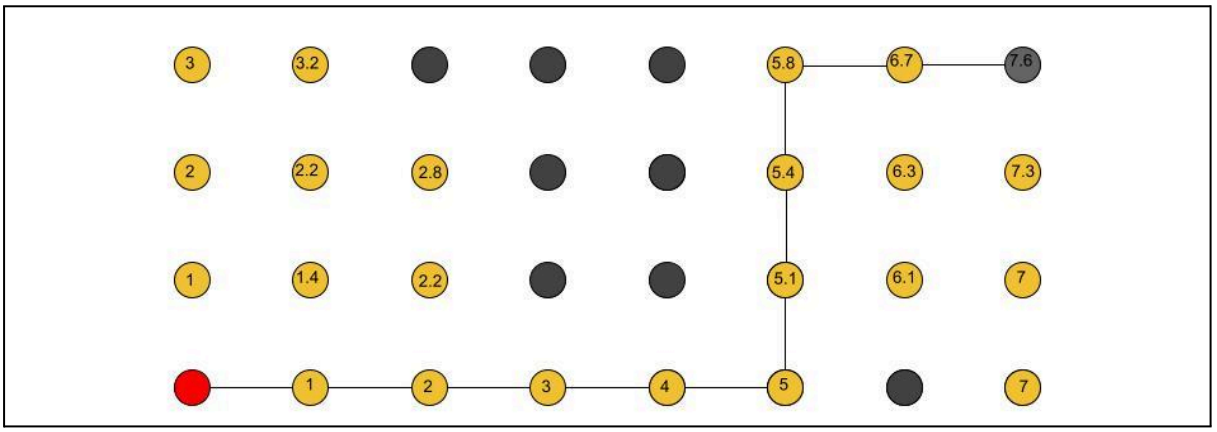


First, the algorithm will prioritize the shortest distances like this:



However, the algorithm relies on the fact that no edges can have a negative cost. Since all edges have a negative cost on this route, the algorithm looks for another route using the

unvisited blocks. Since we already went through the “1 - 1.4” route, we will now try the algorithm on the “1 - 2” route.



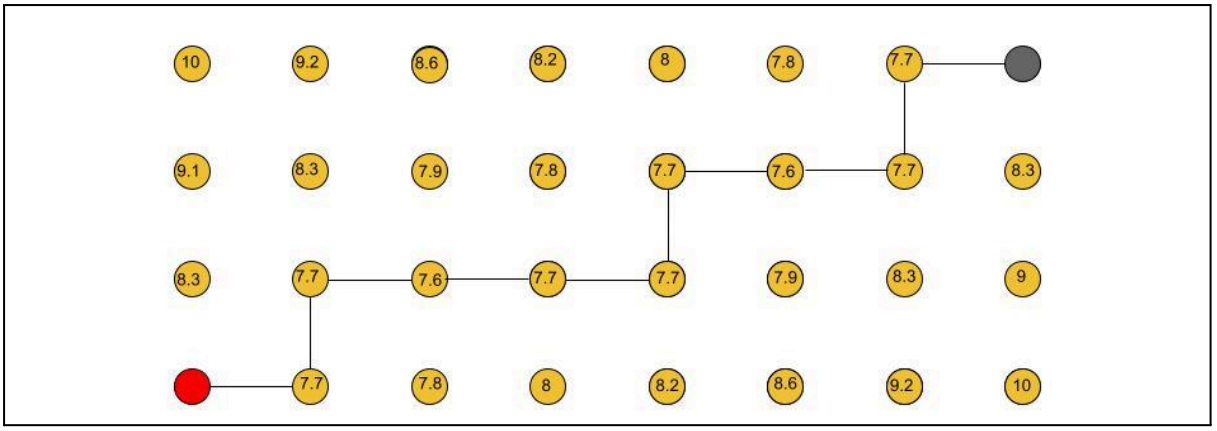
Now, we have found the fastest route. Although the algorithm will always find the fastest route, it requires a lot more time than the Greedy First-Search Heuristic in cases where there are no obstacles because it must check unvisited vertices.

A* Algorithm

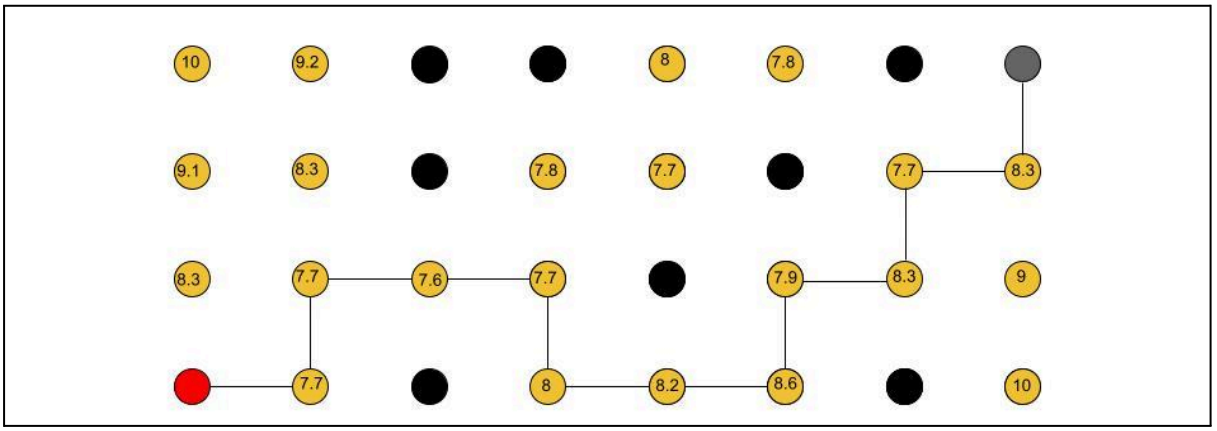
By combining both the speed of the Greedy First-Search Heuristic and the obstacle adaptability of Dijkstra’s algorithm, we get the benefits of both and get rid of the drawbacks for both known as the A* algorithm. Let’s represent the block’s distance from the destination point as “X(n)” and the block’s distance from the origin as “O(n).”

$$C = X(n) + O(n)$$

We add X(n) and O(n) to represent the relative costs of each block to the route because X(n) represents the cost from the destination, and O(n) represents the cost from the origin. Now, we simply follow the blocks with the lowest value to minimize the cost to the path and check the unvisited block when we cannot reach the endpoint.



If we applied only Djkstra's algorithm, it would have taken much more time checking all the unvisited nodes. Here is an application of the A* algorithm on a Minecraft world with blocks that are unreachable.



However, the A* Algorithm does not account for blocks that only slow movement without completely stopping it.

A* Advanced Algorithm

Since the methodology of the A* algorithm follows the blocks that have the cheapest cost to the path, we must add the amount a block slows movement to the cost. We can use this formula to account for the object that slows movement. Let us represent S as the fraction of the normal speed that the player travels in a given block.

$$C = X(n) + O(n) + (1/S) - 1$$

Since each block is 1 meter apart, then the amount that the object slows by represents the additional distance the player must travel. For example, if we move at 25% of our normal speed in a given block, then the cost of that given block to the path must increase by 3 blocks. The time it takes to travel that one block that slows movement, we could have traveled four blocks that don't slow movement.

Movement Slowing Blocks & S Values

Soul Sand: 0.5

Honey Blocks: 0.4

Slime Blocks: 0.4

Cobweb: 0.15

Sweet Berry Bush: 0.34

Water: 0.5

*Ice: 1.05

**Exception: Ice slightly increases player movement which means that it would lower the cost and encourage pathing through ice.*

Just as we used C in the A* Algorithm, we apply it the same way in A* Advanced.

Conclusion

Through our research, we examined various aspects of pathfinding in Minecraft's environment, revealing that minimizing the cost of time and distance from the origin and

endpoint is the most efficient pathfinding. Through the applications of Dijkstra's algorithm, we were able to create a new algorithm to find this efficient pathfinding. This optimization approach does not simply improve speedrunning tactics, but also holds further applications in the pathfinding of terrain that can be represented on a grid. Additionally, the implications of considering time as a cost in the A* Advanced Algorithm may be used in other algorithms of time-based optimization rather than distance-based optimization.

References

Keller, M. T., & Trotter, W. T. (2017). Applied Combinatorics. Mitchel T. Keller, William T. Trotter.

Braam, Isaac. "BlockStep: An A* Algorithm Toward Optimal World Traversal for Speedrunning in Minecraft." Bachelor's thesis, Leiden Institute of Advanced Computer Science (LIACS), Leiden University, 2022. https://theses.liacs.nl/pdf/2021-2022-BraamI.pdf

Velzel, S. (2020). Natural Path Generation: Patch-based terrain optimization for use in games [Master's thesis, Utrecht University]. August 21, 2020. ICA-4091655