3D Drone Navigation Using Augmented Astar Cost Function

Anthony Clark

August 12th, 2025

1 Problem Description

This project addresses autonomous navigation in a constrained 3D environment, modeled as a multi-floor grid-based map. The agent—a simulated drone—must travel from a starting point to a designated goal while navigating obstacles, traversing between floors via hatches, managing limited battery capacity, and avoiding dynamically moving enemies. Such navigation problems arise in real-world settings including unmanned aerial vehicle (UAV) operations in warehouses, multi-story parking facilities, or search-and-rescue missions inside complex buildings.

A baseline A* search algorithm, uses the standard cost function:

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

In this formulation, $g(n)$ represents the actual cost accumulated from the start node to the current node n, while $h(n)$ is the heuristic estimate of the remaining cost from n to the goal. The total cost function $f(n)$ guides the search by balancing the known path cost with the estimated cost to reach the destination. This baseline cost function is capable of finding optimal paths in static 2D grids, however, our problem introduces additional real-world complexities such as energy constraints, recharge stations, and time-dependent hazards. In the unmodified A* formulation, these factors are either ignored or handled indirectly, often resulting in inefficient or unsafe routes.

To address this, we augmented the traditional A* cost function with new terms that explicitly model the problem's constraints and priorities. The augmented cost function integrates battery usage, enemy-avoidance penalties, multi-floor traversal costs, and incentives for recharge station proximity, enabling more intelligent and context-aware route planning.

2 Methods

2.1 Maze Structure and Environment

The navigation environment is modeled as a discrete 3D grid composed of stacked 2D floors. For simplicity, both space and time are quantized: the drone moves in discrete steps between adjacent coordinates, and each action (move or wait) advances the simulation clock by one time unit. All floors share identical 2D dimensions to maintain consistent pathfinding logic, and floor-to-floor movement is achieved through defined hatch coordinates. Enemy drone trajectories must be pre-defined and are re-established whenever a new simulation begins. Sample maps and hatch configurations used in this work are provided in the project's GitHub repository.

2.2 Enemy Drones

Enemy drones follow fixed, repeating trajectories defined on a per-floor basis. At each time step, enemy positions are updated, and any planned drone path intersecting an enemy position at the same time is considered invalid.

2.3 Recharge Stations and Battery Life

The drone has a limited battery capacity, measured in discrete movement units. Recharge stations, marked on the map, instantly restore the drone's battery to maximum when visited, enabling longer traversal and avoiding mission failure due to energy depletion.

2.4 Hatches (Multi-Floor Layout)

Hatches provide vertical connectivity between floors, allowing the drone to move up or down one floor at a time. Each hatch has a corresponding coordinate directly above or below on the connected floor, ensuring consistency in 3D navigation.

Element	**Symbol**	**Description**
Available Space	A	Coordinate on the map with no other elements that either the drone agent or enemy drone can freely move through.
Obstacle	#	Coordinate on the map that neither the drone agent nor the enemy drone can move through; fully occupied space.
Recharge Station	+	Coordinate that restores the drone's battery to maximum capacity when visited.
Enemy Drone	!	Coordinate occupied by an enemy drone at a specific time step, based on its trajectory.
Up Hatch	^	Coordinate where the drone can move up one floor.
Down Hatch	v	Coordinate where the drone can move down one floor.
Goal	$	Target coordinate the drone must reach to complete the mission.

Table 1: Map symbols and their corresponding meanings for each 2D floor.

2.5 Rules of Drone Movement

The drone can move exactly one coordinate per unit of time, either horizontally, vertically within a floor, or vertically between floors via a hatch. Each movement consumes one unit of battery life, regardless of the direction or type of movement. Waiting in place does not consume battery power but still advances the simulation clock by one time unit. Moving through a hatch from one floor to the corresponding coordinate on the connected floor is treated as a standard movement in both time and battery cost.

2.6 Meta Data

Simulation parameters and environment configurations are stored in a JSON file. This file specifies the battery life of the drone, the dimensions of each floor, the number of floors, and the initial and goal coordinates. Each floor is represented as a 2D matrix of symbols as defined in Table **??**. At present, all 2D floors must share the same dimensions to ensure consistency in movement logic and hatch connectivity. The JSON format also includes enemy drone trajectories and the locations of recharge stations and hatches.

2.7 Cost Function

We compare five cost-function variants. Let $g(n)$ be the path cost from start to node n, and $h(n)$ be the heuristic estimate from n to the goal. Let battery_used(n) be the cumulative energy consumed

upon reaching n, floor_distance(n, goal) be the number of floor transitions still required from n to the goal (e.g., $|\text{floor}(n) - \text{floor}(\text{goal})|$ when floors connect sequentially by hatches), enemy_penalty(n, t) be a time-dependent risk term if an enemy occupies or threatens n at time t, and d_recharge(n) be the grid distance from n to the nearest recharge station on the same or reachable floor.

We use nonnegative weights $\beta, \gamma, \varepsilon, \iota$ to tune the relative influence of each augmentation:

cf0: $f_0(n) = g(n) + h(n)$

cf1: $f_1(n) = g(n) + h(n) \ + \ \beta \cdot \text{battery}(n)$

cf2: $f_2(n) = g(n) + h(n) \ + \ \beta \cdot \text{battery}(n) \ + \ \gamma \cdot \text{floor}(n, \text{goal})$

cf3: $f_3(n) = g(n) + h(n) \ + \ \beta \cdot \text{battery}(n) \ + \ \gamma \cdot \text{floor}(n, \text{goal}) \ + \ \varepsilon \cdot \text{enemy\_penalty}(n, t)$

cf4: $f_4(n) = g(n) + h(n) \ + \ \beta \cdot \text{battery}(n) \ + \ \gamma \cdot \text{floor}(n, \text{goal}) \ + \ \varepsilon \cdot \text{enemy\_penalty}(n, t) \ - \ \iota \cdot \dfrac{1}{\text{d\_recharge}(n) + \tau}$

Here $\tau > 0$ is a small constant to avoid singularities when n is on a recharge station. The floor_distance(n, goal) term explicitly encodes vertical progress: if the agent starts on floor 0 and the goal is on floor 4, the term decreases as the path approaches hatches that lead upward, encouraging timely floor transitions. The proximity incentive $-\iota/(\text{d\_recharge}(n) + \tau)$ makes nearby recharge stations relatively more attractive when energy is scarce.

2.8 Example of Map

Several different maps were tested to develop the cost function. Different versions of maps were used to develop terms in the cost function for instance maps with large numbers of recharge stations were used to develop the "distance to nearest recharge term". As it stands, the program can only load in Maps that have floors with the same dimensions. So if the ground floor is 13 x 20, and there are 10 floors in total, then all ten floors need to be 13 x 20 or else the loading element will throw an error. Below is an example of a simple, two-floor map with an agent drone, goal state, recharge stations, obstacles and enemies with fixed trajectories. In this map, each space is represented by a square with blue representing the starting state of the agent drone, red representing the goal state, black representing the obstacles, green representing enemies, gray representing recharge stations and purple representing hatches, which allow the drone to move from one floor to another floor.

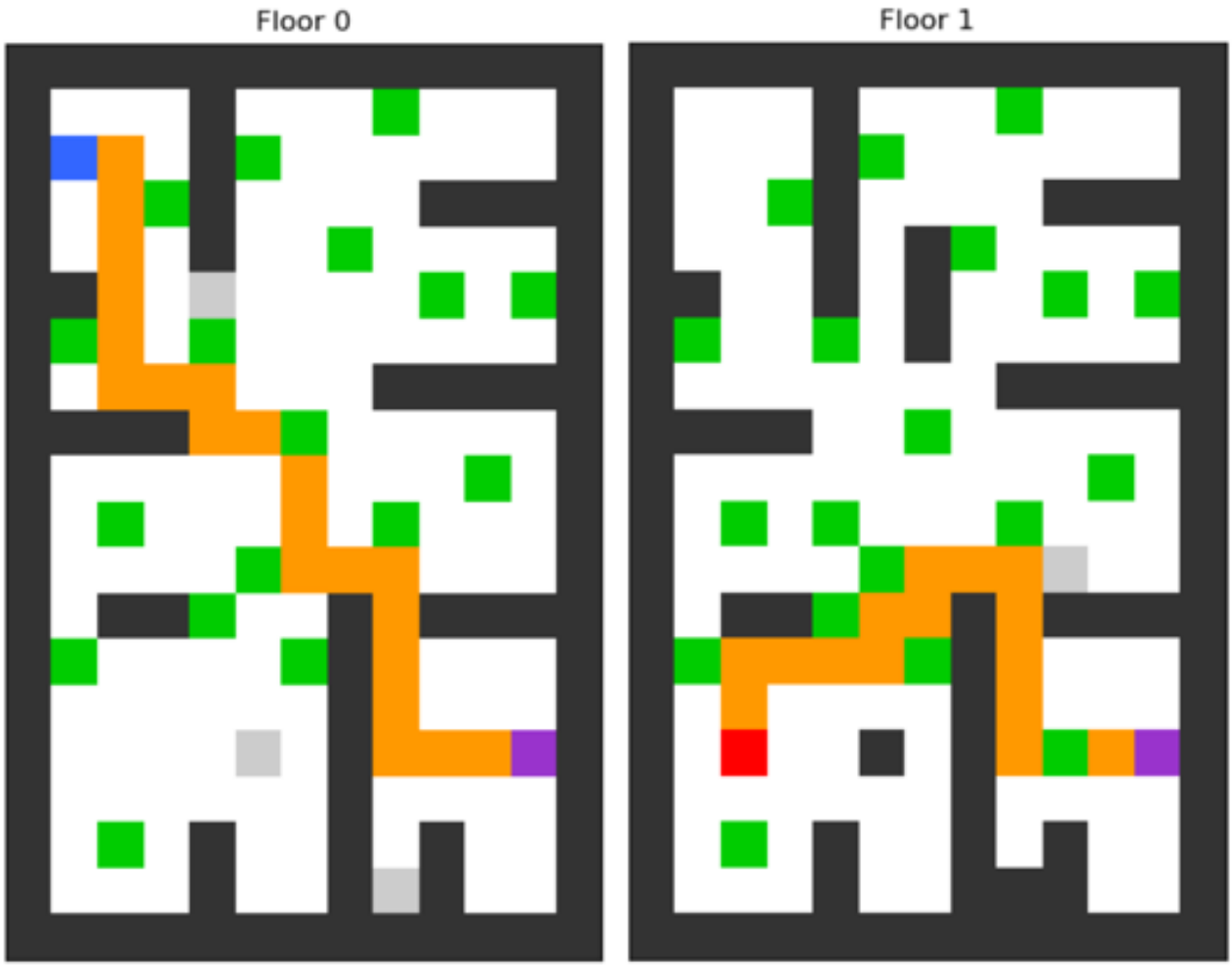

Figure 1: Example of Map (Large amounts of enemies) The drone does not pass through any enemy because the grid only shows the enemy start states, the trajectories are not shown for the sake of visualization

3 Analysis

Our evaluation focuses on how different heuristics and cost function weightings influence the efficiency of path planning, measured primarily by the number of iterations (node expansions) required to discover the optimal path. Since all tested heuristics are admissible, each configuration ultimately produces the same optimal path; however, the computational effort to find that path varies significantly.

3.1 Analysis 1: Term Comparison on a Fixed Map

This analysis compared the performance of five progressively augmented cost functions, all applied to the same fixed 3D multi-floor map. The baseline function, $cf0$, uses only the standard A* formulation $f(n) = g(n) + h(n)$. $cf1$ adds a battery usage term, $cf2$ further incorporates a floor-distance penalty to account for vertical traversal toward the goal, $cf3$ adds an enemy penalty term, and $cf4$ introduces an incentive for proximity to recharge stations. To isolate the impact of these terms without the confounding effects of dynamic hazards, all enemy trajectories were disabled for this experiment. Each cost function was evaluated with both Manhattan and Euclidean heuristics, ensuring that the map, start/goal positions, and other parameters remained constant across runs. The primary performance metric was the number

of node expansions required to find the optimal path, with the final plan steps recorded to verify path consistency.

Heuristic	Cost Function	Expanded Nodes	Final Plan Steps
Manhattan	cf0	74844	158
Manhattan	cf1	74482	158
Manhattan	cf2	59695	158
Manhattan	cf3	59695	158
Manhattan	cf4	59707	158
Euclidean	cf0	71241	158
Euclidean	cf1	71179	158
Euclidean	cf2	70459	158
Euclidean	cf3	70459	158
Euclidean	cf4	69860	158

Table 2: Comparison of cost functions with Manhattan and Euclidean heuristics on the same map, with enemy trajectories disabled.

Results show that Manhattan consistently outperformed Euclidean across all cost functions, expanding fewer nodes while producing the same optimal 158-step path. The addition of the battery usage term ($cf1$) did not provide measurable benefits in this experiment, likely due to the high maximum battery capacity available. The most significant efficiency gain came from adding the floor distance term ($cf2$), which helped guide the search more directly toward the goal floor. The enemy penalty term ($cf3$) had negligible effect in this configuration since enemies were disabled. Finally, the recharge distance term in $cf4$ slightly increased node expansions compared to $cf2$ and $cf3$, as it occasionally prioritized proximity to recharge stations despite the agent's ample battery reserves. These results suggest that, in scenarios with generous battery capacity and no active enemies, vertical distance weighting is the most impactful augmentation to the baseline cost function.

3.2 Analysis 2: Comparison with Enemy Trajectories Enabled

In Analysis #2, we re-ran our planner with **enemy trajectories enabled** to evaluate how different cost functions and heuristics adapt to dynamic hazards. This required several pivotal algorithmic changes: we implemented a **global enemy period** with precomputed occupancy maps to track enemy positions at each time phase, added **edge-swap collision checks** to prevent moving into an enemy's path as it moves into ours, and overhauled **WAIT logic** to occur only when strategically beneficial and to recharge the battery if stationary on a charger. We also integrated **priority-aware dominance pruning** that factors in path cost and heuristic value while using the cost-function terms (battery usage, floor distance, recharge lure) only as tie-breakers. These enhancements ensured the planner could safely navigate around moving threats without excessive waiting or unnecessary expansions.

The results show that enabling cost-function guidance significantly reduced expansions for both heuristics, with all configurations producing the optimal 161-step path. Interestingly, **Euclidean outperformed Manhattan** when enemy trajectories were active, achieving the lowest expansion counts across cf1–cf4. This suggests that in environments with moving hazards, the Euclidean metric's smoother distance estimation better complements the tie-breaker guidance, leading to more efficient exploration. The consistent step counts confirm that these gains were achieved without sacrificing path optimality.

3.3 Analysis 3: Multi-Floor while varying gamma

We evaluated a range of floor-distance weights γ across ten maps (Map3–Map12) that vary only in the *number of floors* (1→10). Every floor is 13×20 and follows a shared template, but each map includes

Heuristic	Cost Function	Expanded Nodes	Final Path Steps
Manhattan	cf0	62,574	161
Manhattan	cf1	42,290	161
Manhattan	cf2	42,295	161
Manhattan	cf3	42,295	161
Manhattan	cf4	42,295	161
Euclidean	cf0	68,337	161
Euclidean	cf1	40,284	161
Euclidean	cf2	40,284	161
Euclidean	cf3	40,284	161
Euclidean	cf4	40,284	161

Table 3: Performance with enemy trajectories enabled across heuristics and cost functions.

slight internal variations (obstacles, hatch placement) to avoid overfitting to a single layout. We disabled enemies to isolate γ's effect and held all other settings fixed while using `cf2` (battery + floor) with the Euclidean heuristic. Here, γ controls how strongly the planner prefers reducing vertical separation $|\Delta z|$: a larger γ increases the additive cost-pressure to reach the correct floor sooner, biasing the search toward earlier hatch transitions while leaving path optimality (steps) unchanged.

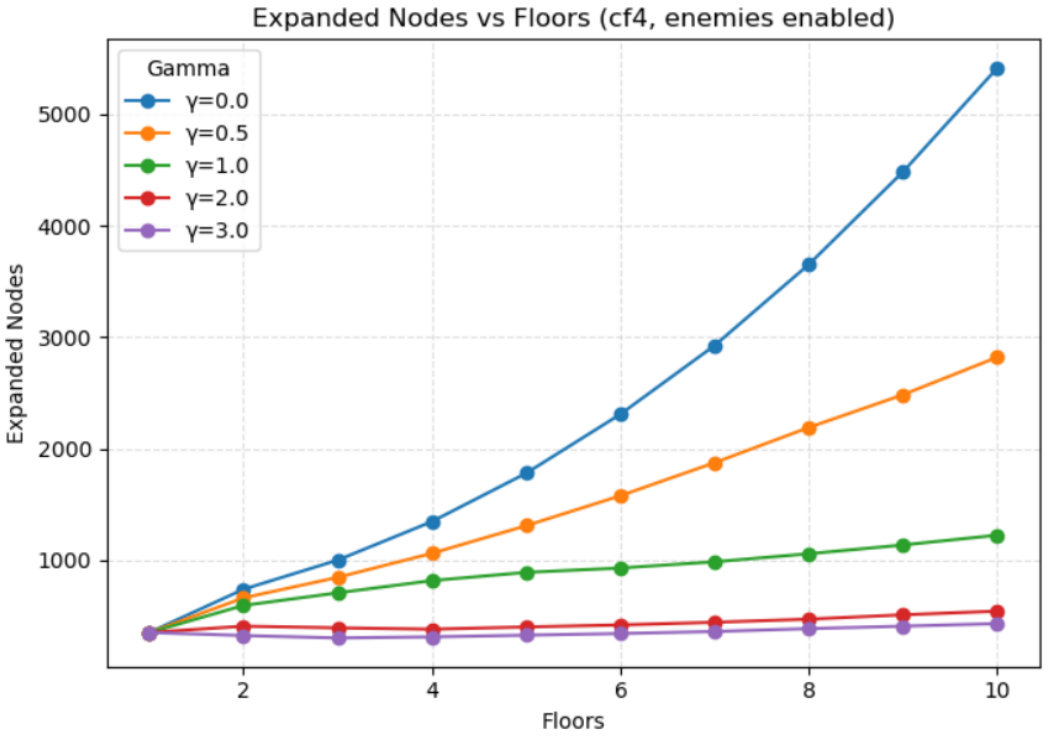

Figure 2: Number of expanded nodes vs number of floors in map. Varying gamma range (0.0, 0.5, 1.0, 2.0, 3.0)

We observe a large reduction in expanded nodes from $\gamma = 0 \rightarrow 0.5$, and another big drop from $0.5 \rightarrow 1.0$ across multi-floor maps. The improvement from $\gamma = 1.0 \rightarrow 2.0$ is smaller but still substantial, and from $2.0 \rightarrow 3.0$ it tapers further—yet $\gamma = 3.0$ remains the best whenever the map has more than one floor. As expected, the single-floor case shows no change (no vertical distance to reduce). Overall, increasing γ helps the planner commit to the target floor earlier, trimming lateral detours without altering

the final path length.

3.4 Analysis 4: Multi-Battery-Lives While Varying Recharge Weights

For this figure we examine how recharge shaping affects search effort on Map13 (two floors, 30×45, no enemies). We vary the recharge lure weight ι and the smoothing term τ while holding all other settings fixed, and plot the number of expanded nodes versus ι, with a separate line for each battery capacity. In our cost shaping, the recharge term enters as $-\iota/(d_{\text{re}} + \tau)$, where d_{re} is the grid distance to the nearest charger. Intuitively, ι controls *how strongly* the planner is pulled toward chargers (larger ι means a stronger attraction), while τ acts as a small stabilizer that limits the peak attraction when d_{re} is very small and gently smooths the lure's gradient. We sweep $\iota \in \{0, 0.5, 1, 2, 4\}$, $\tau \in \{10^{-6}, 10^{-4}, 10^{-3}, 10^{-2}\}$, and battery levels $\{38, 48, 57, 76\}$, where 38 is the minimax-feasible capacity for this map.

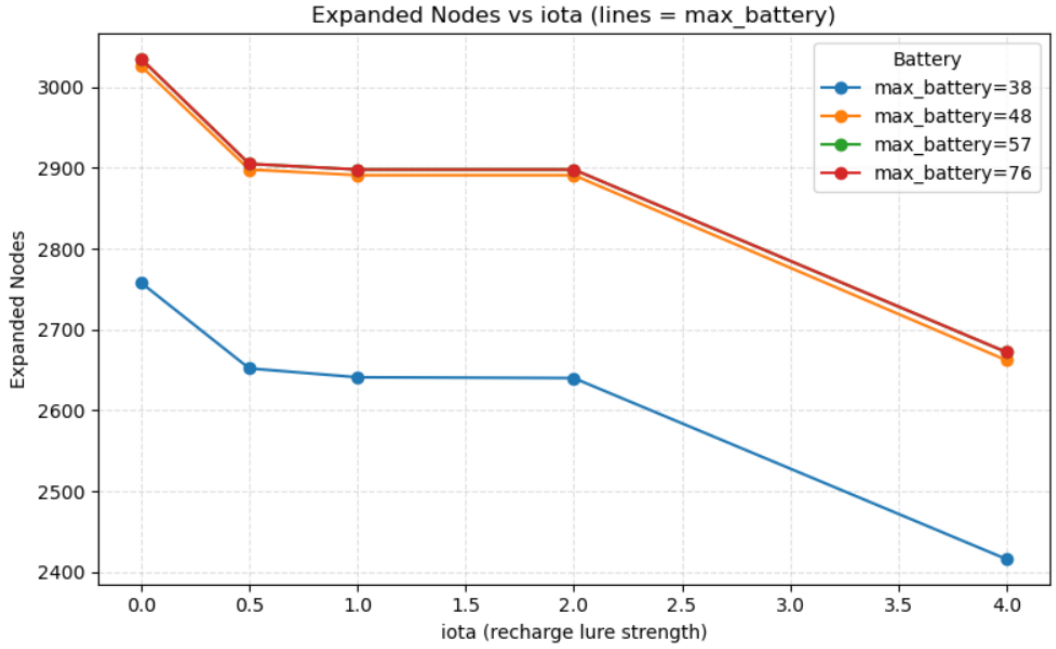

Figure 3: Number of expanded nodes vs values of iota (0 through 4.0) with different max battery lives (38, 48, 57 and 76)

Across the grid, varying ι substantially changes the number of expanded nodes, whereas varying τ has little visible effect. This is expected: along typical routes d_{re} is on the order of several cells, so adding $\tau \in [10^{-6}, 10^{-2}]$ barely alters $1/(d_{\text{re}} + \tau)$, while scaling by ι directly shifts the trade-off between detouring to a charger and pushing forward. As ι increases, expansions consistently drop, with $\iota = 4$ yielding the strongest reductions at all battery levels. We also see that the max_battery $= 38$ curve sits markedly lower overall—near the feasibility threshold the search space is tighter and the lure quickly commits to charger-supported corridors—whereas max_battery $\in \{48, 57, 76\}$ produce very similar curves across all ι, reflecting that ample capacity admits many near-equivalent routes and thus more uniform search effort.

4 Future Work

There are several promising directions to expand this project beyond its current scope. First, larger and more complex maps will be incorporated to test the scalability and efficiency of the augmented A* algorithm in environments that more closely resemble real-world navigation challenges. Future iterations will also support levels with varying 2D floor dimensions, such as transitioning from a 14×8 floor to a

20×13 floor, requiring more flexible pathfinding logic and hatch mapping. Further refinement of the cost function is planned, particularly optimizing the balance of battery life prioritization to ensure it neither overly penalizes nor underestimates energy management in path selection. Additionally, more advanced planning algorithms—such as D*, Anytime Repairing A*, or multi-agent planning frameworks—could be integrated to improve adaptability in dynamic environments. To diversify test cases, a procedural generation module will be developed to create randomized maps and enemy trajectories, ensuring that the algorithm is robust across a wide range of scenarios. Enemy drone behavior will also be enhanced to include pursuit programs, enabling them to actively track and intercept the player's position rather than simply following pre-set routes. Battery consumption modeling will be made more realistic by introducing a partial penalty for waiting; although less costly than movement, waiting will still consume some battery life, discouraging excessive idling. Finally, dynamic elements such as moving obstacles and mobile goals will be added to simulate real-time mission changes, pushing the algorithm toward more reactive and adaptive decision-making. Collectively, these enhancements will deepen the complexity of the navigation problem, improve the realism of the simulation, and broaden the applicability of the system to real-world autonomous drone operations.