University Carlos III of Madrid

BSc in Applied Mathematics and Computing

Artificial Intelligence Project

Member 1: Sabela Rubert Docampo
100523102
100523102@alumnos.uc3m.es

Member 2: Roberto Hogas Goras
100523139
100523139@alumnos.uc3m.es

Member 3: Alejandro Carrascosa Gordo
100523032
100523032@alumnos.uc3m.es

Academic Year 2024/25 Group 121

With love, made in LaTeX for Vidal, our Artificial Intelligence practice teacher.

Click here to see our Overleaf code.

Contents

1 Introduction **2**

2 Main Content **2**
2.1 Problem Modeling 2
2.1.1 States and Operators 2
2.1.2 Initial and Goal States 3
2.1.3 Admissible Heuristics 3
2.2 Map Implementation 4
2.3 Search System Implementation 4

3 Experiments **5**

4 Use of Generative AI **13**
4.1 Aspects in which AI was used 13
4.2 How AI has facilitated the completion of the project 13
4.3 Verification and adaptation of the information obtained 13

5 Conclusion **14**

1 Introduction

In this project, we were asked to work on finding a safe path for a stealth plane that needs to fly over a determined grid and take pictures of several important areas without getting detected by radar. Even though the plane uses stealth technology, it's not completely invisible, so we have to be careful about where it flies. To help with this, we use a detection probability map that shows how risky each part of the area is. Our goal is to plan a route using heuristic search that lets the plane visit all the required points while avoiding high-risk zones and keeping radar exposure as low as possible. This project is sectioned in three big main parts: the problem modeling, the map implementation and the search system implementation. Additionally, there's an experiments section where we test the different scenarios provided with the project statement. Finally, a section to discuss the use of generative AI and how we adapted the information and some final thoughts in the conclusion.

2 Main Content

2.1 Problem Modeling

2.1.1 States and Operators

States:

A state $s \in S$ is defined as a cell in the discretized radar exposure grid of size $H \times W$, represented by its grid coordinates:

$$s = (i, j), \quad \text{with} \quad i \in \{0, \ldots, H-1\}, \quad j \in \{0, \ldots, W-1\}$$

The state space is restricted to grid cells whose exposure value does not exceed a threshold:

$$S = \{(i, j) \mid \Psi^*_{\text{scaled}}(i, j) \leq \theta\}$$

where Ψ^*_{scaled} is the normalized radar exposure value and $\theta \in (0, 1]$ is the maximum allowable detection risk.

Operators:

The agent (aircraft) can move to orthogonally adjacent cells. The set of actions is:

$$A = \{\text{Up}, \text{Down}, \text{Left}, \text{Right}\}$$

The transition function $\delta(s, a)$ is defined as:

$$\delta(s, a) = \begin{cases} s' & \text{if } s' \text{ is adjacent, in-bounds, and } \Psi^*_{\text{scaled}}(s') \leq \theta \\ \text{undefined} & \text{otherwise} \end{cases}$$

Cost Function:

The cost of moving from state s to a valid neighboring state s' is given by:

$$\text{cost}(s, s') = \Psi^*_{\text{scaled}}(s')$$

This encourages the search to favor paths through cells with lower exposure.

2.1.2 Initial and Goal States

Initial State:

$$s_0 = \text{POI}_1$$

The initial state is the grid cell corresponding to the first point of interest (POI) in the mission.

Goal State:

$$s_g = \text{POI}_k \quad \text{with } k > 1$$

Each local planning subproblem uses the next POI as a goal. The global objective is to visit all POIs in the specified sequence, chaining together the paths obtained by running A* between consecutive POIs.

2.1.3 Admissible Heuristics

Heuristic 1: Euclidean Distance

$$h_1(s, s_g) = \sqrt{(i - i_g)^2 + (j - j_g)^2}$$

This heuristic underestimates the true cost because it ignores radar exposure. It is efficient and intuitive in dense or continuous environments.

Heuristic 2: Manhattan Distance

$$h_2(s, s_g) = |i - i_g| + |j - j_g|$$

This is admissible and consistent when only orthogonal movements are allowed. It tends to be more accurate than Euclidean distance in grid environments with constrained motion.

Admissibility Justification:

Both heuristics are admissible because:

- They never overestimate the true minimal cost between two states.
- They represent the minimum number of steps to reach the goal under ideal (zero-cost) conditions.
- Since each step incurs at least a minimal exposure $\varepsilon > 0$, the true cost is always greater than or equal to the heuristic estimate.

2.2 Map Implementation

The **map** is generated as a **discrete grid covering a real-world area defined by geodetic coordinates**. Using the `Map` class, we create a grid of size $H \times W$, where **each cell** contains **latitude** and **longitude values**. **Radars** are randomly placed within this grid, and their properties (e.g., power, gain) are set to compute their **maximum detection range** using the **radar equation** (Equation 1).

For each cell, we calculate the **detection probability** using a **multivariate Gaussian distribution** (Equation 2). If a cell is within a **radar's detection range**, its detection value is updated to the **maximum value** from **all radars** (Equation 3). These values are then scaled using `MinMax` **normalization** (Equation 8) to ensure they fall within $[\varepsilon, 1]$, **avoiding zero probabilities**. This is handled in the `compute_detection_map` method.

Search Space Graph Construction:

The **search space** is represented as a directed graph using `networkx.DiGraph`. The **detection map** is processed by the `build_graph` function, which adds edges for legitimate transitions (left, right, up, and down) and **nodes** for **every cell**. Only when the **detection probability** of the **adjacent cell** is less than a **specified tolerance** threshold is a transition permitted.

By allocating **edge weights** based on the **destination cell's detection probability**, the **A* algorithm** makes sure that **safer routes** are given **priority**. For example, moving from cell A to B assigns a weight equal to B's detection value. By excluding cells that exceed the tolerance, the spy plane is kept out of high-risk areas.

2.3 Search System Implementation

Now that we have already gone into the details and explanations of the code programmed to implement the map, we can go through the implementation of the **search engine based** on the **A* algorithm**. But first, let us go through all the information we have and are going to take into account. We need to **find a path given a certain map**, where **some cells** are considered **obstacles** once they **surpass the given tolerance**. What is more, we are given some specific points called **POI (Points Of Interest)** with some coordinates which we must visit. Our goal when implementing this search engine is to find always the shortest path, and that is when the **A* algorithm** comes in handy. This algorithm will help us find the **shortest path**, going through all POIs and avoiding the cells with a higher chance of being detected by the radar.

First of all, the function `build_graph(detection_map, tolerance)` **turns the detection map into a graph**, where every possible **visiting cell** (the ones with lower tolerance than the established one) turns into a **node**. Then, every safe cell is connected to its neighbouring ones provided they are safe too. Every edge is given a cost. This **cost** is determined by the **chances of being detected in the destination cell**. This all means that moving from the current cell to a more dangerous one costs more than moving to a safer one.

We also need to determine the **coordinates** of our **POIs**, therefore, we use `discretize_coords (coords, lat_bounds, lon_bounds, shape)`. This turns the latitude and longitude coordinates into coordinates in our specific map. With this, our **POIs have a concrete row and column coordinate in our map**.

In order to make this search as efficient as possible, we define **two different heuristic functions**. As we referenced in section 2.1 when modeling our mathematical heuristics, we mentioned the **Euclidean distance** (a straight line) and the **Manhattan distance** (following the grid). We name `h1` the **Euclidean** one, and `h2` the **Manhattan** one. These functions estimate how far the current cell is from our goal cell, which helps the efficiency of A* because they help it prioritize the paths to follow. Each time we use any heuristic, a global counter called `NODES_EXPANDED` increments by one, helping us keep track of **how many nodes we have already evaluated**.

Next, we use `create_visiting_order(pois, graph, heuristic)` to determine the **order of visitation of POIs**. The strategy used is as follows: it starts in the **first POI** and from there it always goes to the **nearest one** (calculated with heuristics). This order is not always the most optimal one, but it is an efficient and fast way of calculating it.

The main part of the answer to the statement lies in the function `path_finding(graph, pois, heuristic_function)`, which **connects** each of the **ordered POIs** using `nx.astar_path`. This function uses the said **A* algorithm**, which now finds the **shortest path between two points**, taking into account all of the previously explained aspects, like the cost of the path and the estimated heuristics in order to prioritize the most promising paths. All of the best paths found are then concatenated to formulate our solution.

Finally, **once the path is all formed**, the function `compute_path_cost(graph, path)` **sums all** the **edges' costs** between the path cells to obtain the **whole cost of the resulting path.**

3 Experiments

First, we tested several **default scenarios** pre-built in `scenarios.json`, each with a **different tolerance** and **heuristic**. Then, we created new custom scenarios to evaluate edge cases and extreme configurations. All scenarios are accompanied by visual graphs, and a summary table is provided for comparative analysis. **Remember that the red dots represent the Points of Interest (POIs), and the blue lines represent the path connecting these POIs.**

scenario_0

- Tolerance used: 0.5
- Number of POIs: 2
- Heuristic used: Euclidean distance
- Total path cost: 3.8112

- Number of expanded nodes: 32
- Description: A basic test with 2 points of interest (POIs) to verify the default behavior of the algorithm.

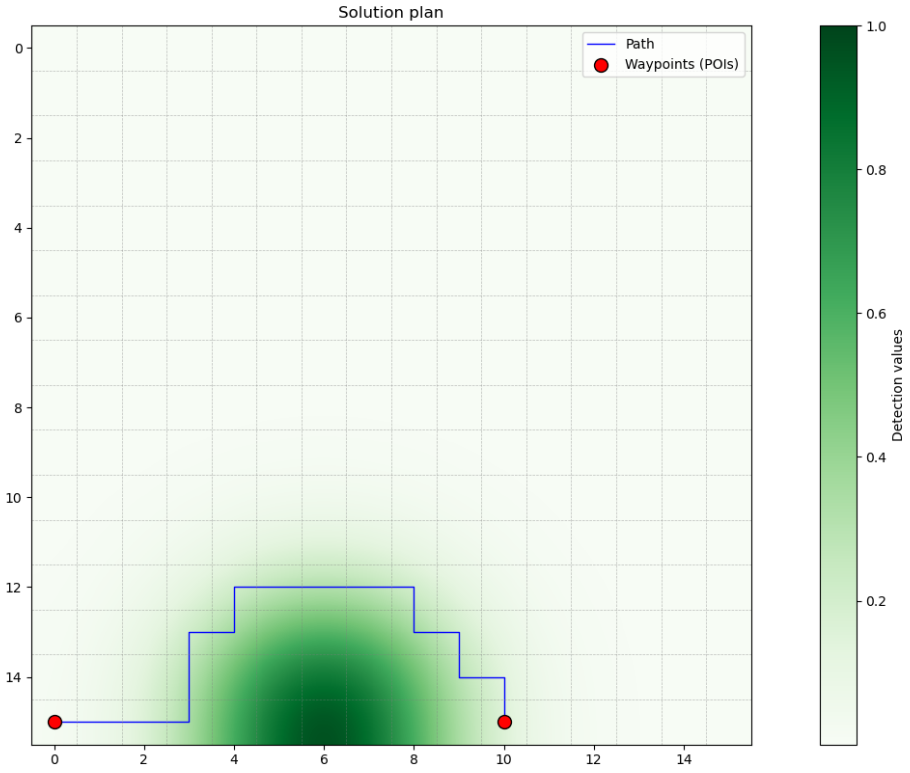

Figure 1: scenario_0 with tolerance of 0.5

scenario_1

- Tolerance used: 1
- Number of POIs: 2
- Heuristic used: Manhattan distance
- Total path cost: 3.3536
- Number of expanded nodes: 22
- Description: A test with 2 POIs and maximum tolerance. As expected, the shortest path is a straight line.

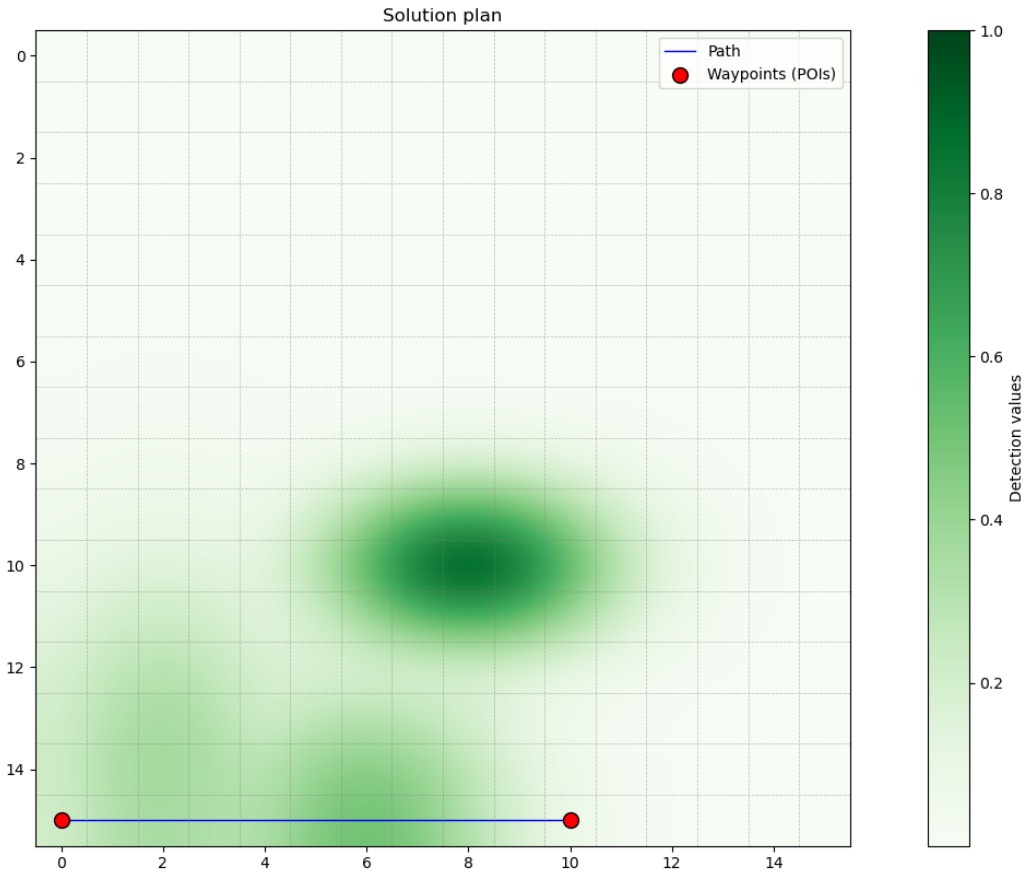

Figure 2: scenario_1 with tolerance of 1

scenario_6 (1st run)

- Tolerance used: 0.25
- Number of POIs: 6
- Heuristic used: Euclidean distance
- Total path cost: 41.8772
- Number of expanded nodes: 756
- Description: A test with several POIs and a very low tolerance. Further decreasing the tolerance may lead to unsolvable paths.

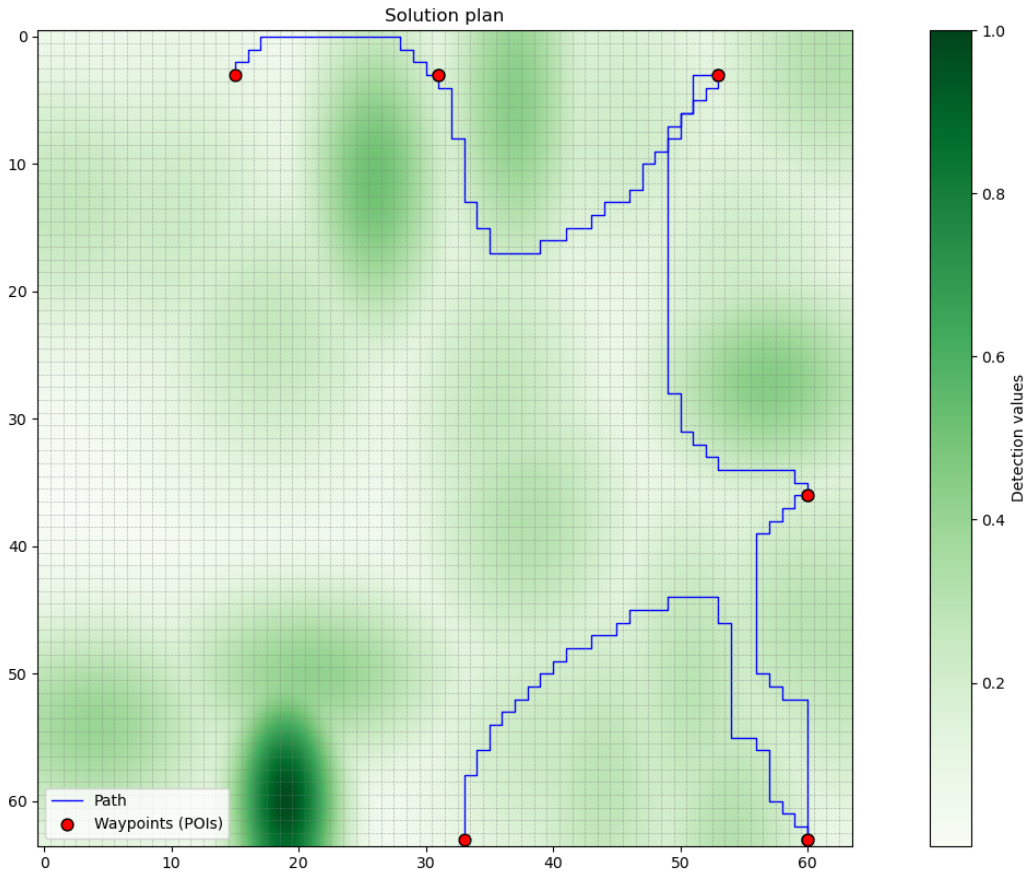

Figure 3: scenario_6 with tolerance of 0.25

scenario_6 (2nd run)

- Tolerance used: 0.1
- Number of POIs: 6
- Heuristic used: Manhattan distance
- Output: POIs to visit: [63, 33], [63, 60], [36, 60], [3, 53], [3, 31], [3, 15]
- ERROR: One of the POIs ((63, 33) or (63, 60)) is not in the graph. The POI may be outside the map or isolated due to the low tolerance.
- Description: With this tolerance level, there is no possible path between some POIs.

scenario_9

- Tolerance used: 0.9
- Number of POIs: 20
- Heuristic used: Euclidean distance

- Total path cost: 4780.5654
- Number of expanded nodes: 69,497
- Description: A stress test with a 1024 × 1024 grid (over 1 million cells). Due to the scale and the high number of radars and POIs, the computational demand is very high and took nearly 4 minutes to process.

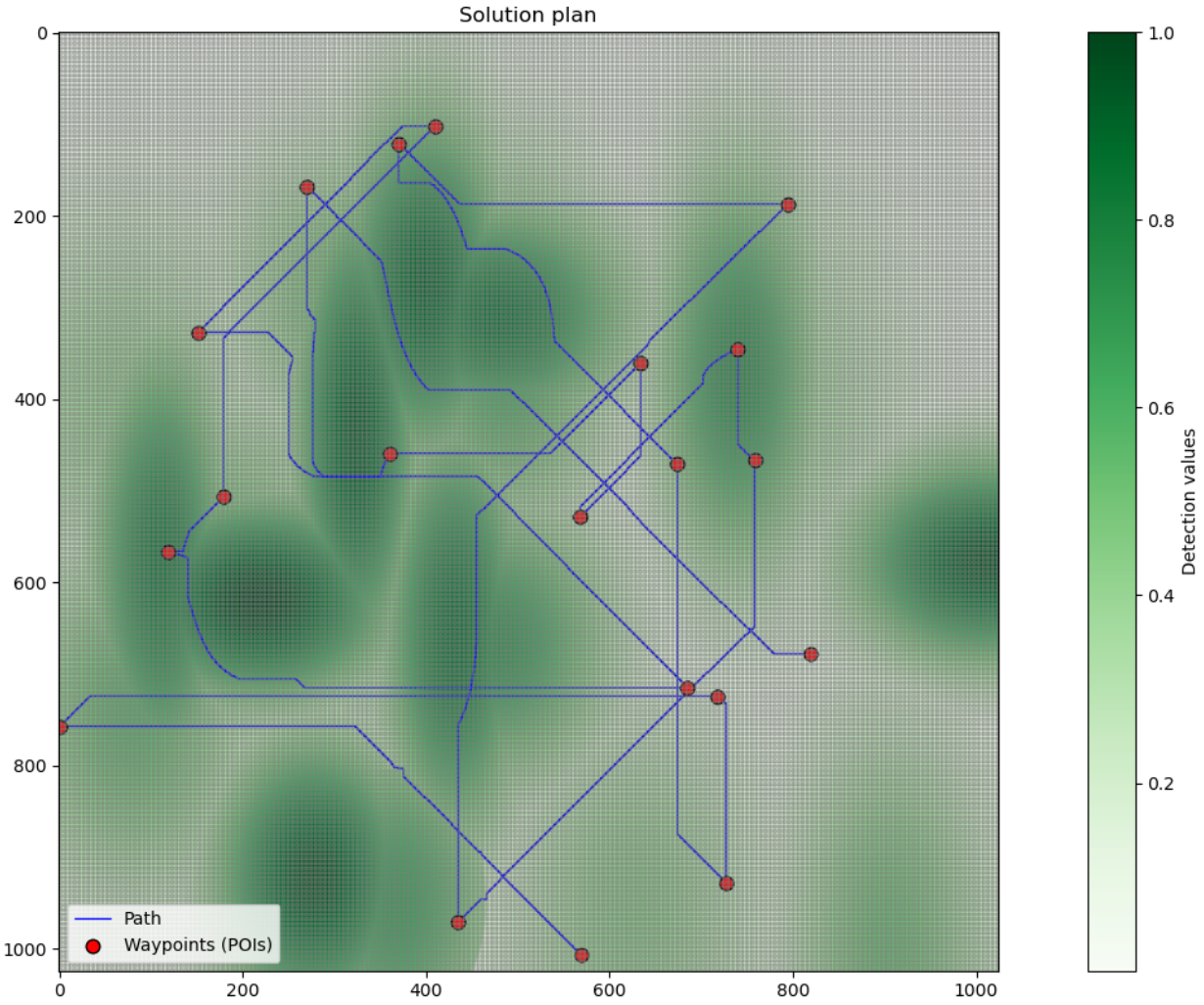

Figure 4: scenario_9 with tolerance of 0.9

Next, we created new scenarios to test additional cases (all of them are included in the `scenarios.json` file, so anyone can test them):

scenario_10

- Tolerance used: 0.5
- Number of POIs: 2
- Heuristic used: Manhattan distance
- Output: POIs to visit: [-3, -586], [-4, -556]
- ERROR: One or more POIs are outside the grid boundaries, causing a failure.
- Description: The POIs are located outside the map, resulting in an immediate error.

scenario_11

- Tolerance used: 0.5
- Number of POIs: 1
- Heuristic used: Euclidean distance
- Output: POIs to visit: [1, 3]
- ERROR: The number of POIs is less than 2, preventing path generation.
- Description: A minimum of two POIs is required to compute a path.

scenario_12 (with Manhattan distance and NO radars)

- Tolerance used: 0.5
- Number of POIs: 2
- Heuristic used: Manhattan distance
- Total path cost: 0.0007999999797903001
- Number of expanded nodes: 12
- Description: A test on a very small grid ($5 \times 5 = 25$ cells) with **no radars**. Since no radar coverage exists, all cells have minimal cost, so the heuristic dominates the pathfinding behavior.

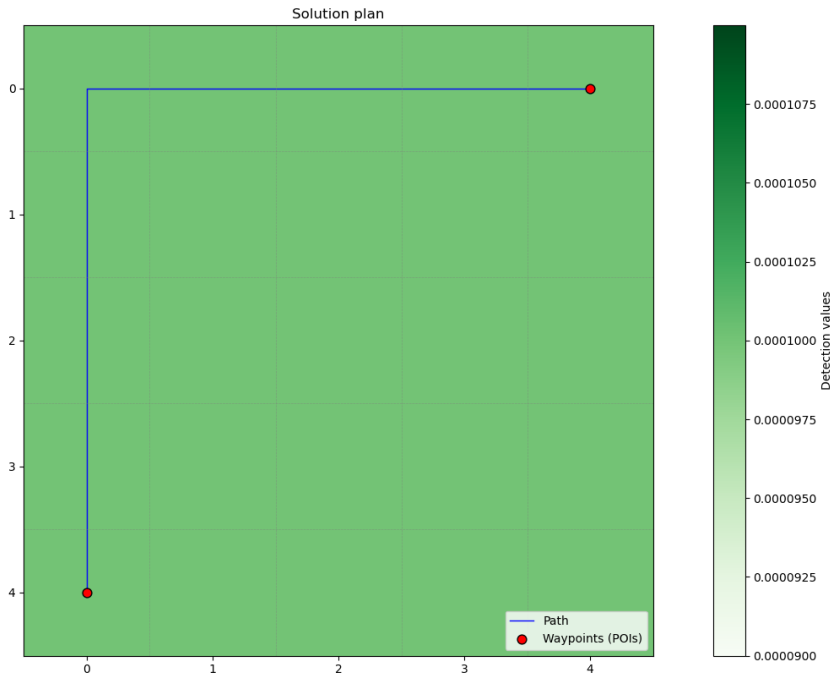

Figure 5: scenario_12 with tolerance of 0.5, no radars and Manhattan distance

scenario_12 (with Euclidean distance and NO radars)

- Tolerance used: 0.5
- Number of POIs: 2
- Heuristic used: Euclidean distance
- Total path cost: 0.0007999999797903001
- Number of expanded nodes: 17
- Description: Same as the previous scenario, but using the Euclidean distance for comparison.

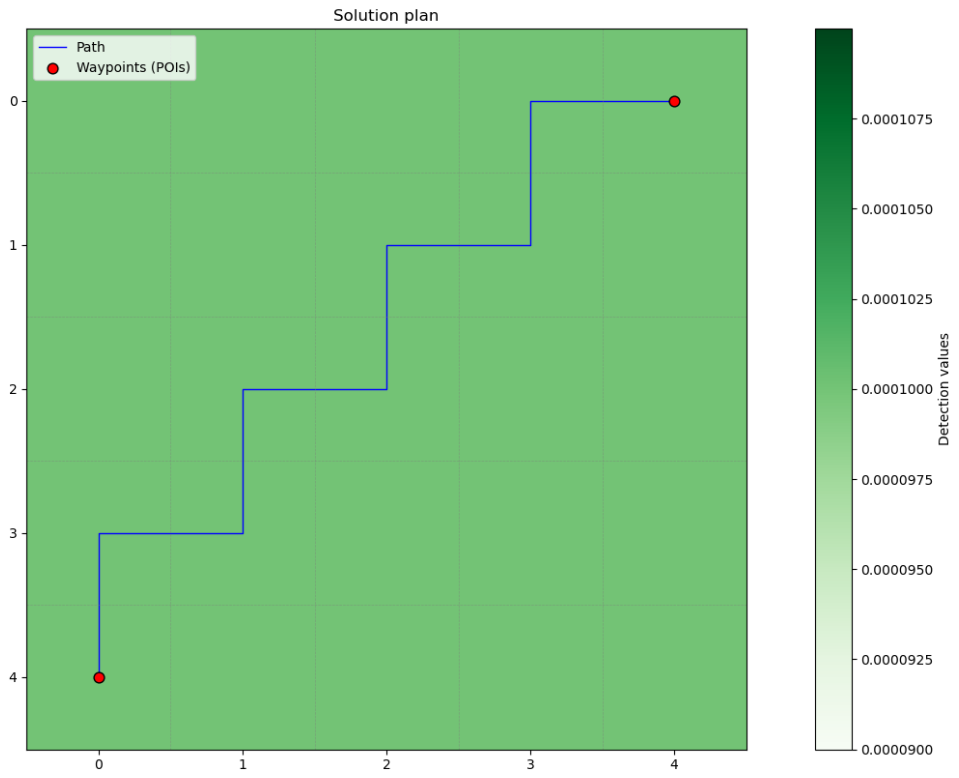

Figure 6: scenario_12 with tolerance of 0.5, no radars and Euclidean distance

Summary table of experiments

Scenario	Tol.	Heuristic	Cost	Nodes	Comment
scenario_0	0.5	Euclidean	3.8112	32	Basic test, 2 POIs
scenario_1	1	Manhattan	3.3536	22	Max tolerance, straight line
scenario_6 (1st)	0.25	Euclidean	41.8772	756	Very low tolerance, many POIs
scenario_6 (2nd)	0.1	Manhattan	–	–	No path because of tolerance
scenario_9	0.9	Euclidean	4780.5654	69497	Huge grid, 20 radars/POIs
scenario_10	0.5	Manhattan	–	–	POIs out of bounds
scenario_11	0.5	Euclidean	–	–	Only 1 POI, no path possible
scenario_12 (1st)	0.5	Manhattan	0.0008	12	Small grid, no radars
scenario_12 (2nd)	0.5	Euclidean	0.0008	17	Same grid, different heuristic

Table 1: Summary of experimental scenarios with tolerance, heuristic, cost, and node expansion

4 Use of Generative AI

4.1 Aspects in which AI was used

While working on the justification for our heuristics, we used generative AI to help us better understand why **Euclidean** and **Manhattan distances** are admissible in our context if they were indeed admissible. The AI helped explain how these **heuristics provide a lower bound** for the actual cost when each move has a small radar exposure value $\varepsilon > 0$. This made it clearer to us why **these functions don't overestimate the true cost** and how they relate to the theory we saw in class and that is stated in the presentations used in lectures. We didn't copy the explanation directly, but used it as a guide to write our own version in a way that made sense for our problem.

In addition, we used AI tools to clarify the usage of key Python functions. Especially we had to look up those related to the `networkx` library, heuristic definition and **A* search**. This included understanding the correct number and type of arguments expected by each functions and taking into consideration the impossible situations for **error handling**. For instance, when testing **edge cases** such as **empty graphs** or **unreachable nodes**, the AI provided insight into how the implementation might behave and whether exceptions needed to be caught or conditions imposed.

4.2 How AI has facilitated the completion of the project

AI was a really helpful technological tool during the development of our project, especially when we were trying to understand some of the more **complex theoretical ideas** or **programming strategies**. For example, when we came across **Python** functions we did not know how to manage or weren't sure why an error was happening, we used AI to get quick explanations that were specific to our problem. This always made things easier to grasp than just trying to figure it out with the theory or presentations. **It helped us better understand how to adjust heuristics into our algorithm for our specific statement**.

AI also helped us write **cleaner** and sometimes more **optimal code**. We asked for advice on how to manage some programming situations and followed good examples so that we could understand easier. We didn't just copy what it said, instead, we used the suggestions to learn and improve our own implementation. And most of the times correcting mistakes in our code.

Overall, **AI** saved us time and **helped us avoid** a lot of **trial-and-error**, which meant we could focus more on designing and testing our algorithm. We always made sure to review and adapt anything we got from AI, so it really acted as a help and not something that replaced our own thinking or work.

4.3 Verification and adaptation of the information obtained

We put in place a rigorous verification procedure to guarantee the accuracy of data derived from AI. First, we compared the AI's explanations of the **Manhattan** and **Euclidean dis-**

tances (two admissible heuristics) with the course materials and hands-on activities. When working with non-negative edge weights, this verification confirmed that these **heuristics never overestimate actual costs**, which is exactly what our radar detection problem requires.

We conducted empirical testing by running controlled experiments on simple grid configurations with known optimal paths. These tests validated that:

- The **Euclidean distance heuristic** (h_1) consistently produced paths that **were equal to or shorter than actual costs**.
- The **Manhattan distance heuristic** (h_2) performed as expected for **grid-based movement patterns**.

All tests were executed using the project's provided base code to ensure compatibility.

For additional validation, we:

1. Consulted Python's official documentation regarding `networkx`'s A* implementation.
2. Discussed AI-generated insights with team members.
3. Sought clarification from our course instructor on specific concepts such as heuristic admissibility.

We meticulously tailored general AI explanations to our particular situation by:

- **Changing heuristic arguments** to take radar detection probabilities into consideration rather than unit costs.
- **Ensuring that all project guidelines**, including **tolerance** thresholds and **POI visitation orders**, were followed in the final implementation.

We made sure all AI-derived data was correct and appropriately contextualized by **combining theoretical validation**, **real-world testing**, and **expert consultation**. No information was included unless it had been critically examined and suitably modified to meet the particular needs of our project. We were able to use AI support while upholding academic integrity and technical accuracy thanks to this meticulous approach.

5 Conclusion

This project has taught us a lot about how to model any problem step by step, using an initial and a final state. Thanks to this modeling, it is much easier to program an algorithm that follows the parameters used to find an appropriate solution. In this project, we used the **A* algorithm**, a well-known algorithm for finding the shortest path between two locations. To do this, we used a **cost function** and a **heuristic function**. Programming both functions was very interesting for us, as we were able to find a real-life application for this type of algorithm (in this case, an airplane that is overexposed to radar).

We think we spent about **30 hours on the practice**, which seems like an adequate amount of time given the workload of programming the functions, understanding the statement,

and knowing what each function already programmed in the code does. The only thing we think was wrong was doing the practice in **Python**, as it is known to be one of the slowest programming languages and explodes when it uses a large amount of data. For example, in scenario 9, **Python** takes over a minute to generate the shortest path graph, while in **Java** or **C++** it would likely take less than a second.

However, overall, we think the project was very interesting and we would do it again if we had the opportunity.