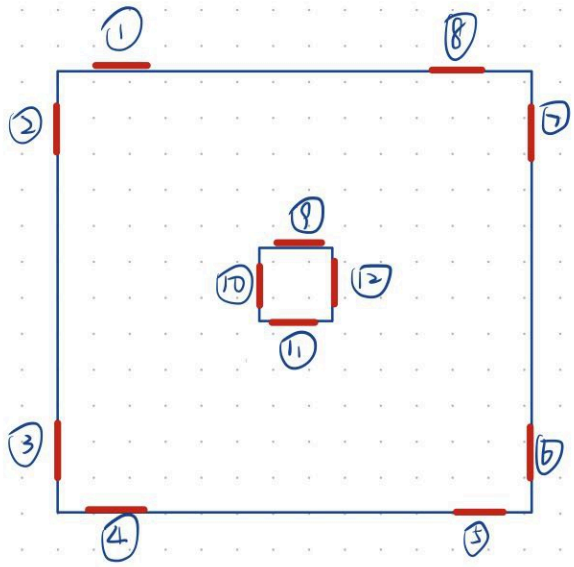
ROBOTICS : 276A HW4 REPORT

PID1: A69034019 Name: Anandhini Rajendran

PID2: A69033245 Name: Yueqi Wu

Environment

We set up a 3m * 3 m(Approx) environment with an obstacle of size 0.5m * 0.5m at the centre. The whole setting including where the landmarks are placed are shown in the following image



Representation

We considered the following representations:

1. Grid representation - it is impractical for large applications but easy to represent and code.
2. Potential field representation - following the lecture slides, we can see that the graph solution is simple, but the algorithm has a chance to fail.

We decided to use grid representation for both the maximum safety task and the minimum time/distance task for the following reasons:

1. A grid divides the environment into uniform cells, making it easy to represent and visualize the world. Each cell can be marked as a free space or an obstacle. This binary representation is straightforward and easy to manage programmatically.
2. Many path-planning algorithms like A* and Dijkstra are designed to work well with grid-based maps. These algorithms can easily expand paths from one cell to adjacent cells based on cost or heuristics. The uniformity of grids simplifies the computation of costs, neighbours, and traversal paths.

3. Grid representation allows straightforward collision checks by marking cells within the robot's radius or near obstacles as occupied or unsafe. This ensures that the robot avoids unsafe zones by assigning high costs or prohibiting traversal through these cells.

Searching/Planning Algorithm

1. Maximum Safety - Dijkstra algorithm
   This algorithm is under the Exact methods in path planning representations. To maximise the safety of the path, we used Dijkstra to plan the route. Dijkstra's algorithm assigns a cost for reaching every point in the grid and minimizes the cost from the start to the goal when planning the path. In this algorithm, we ensure safety by giving the robot a radius variable so that it is not very close to the obstacle.
   - Initialization
     - Resolution: In the grid representation the resolution determines the size of each cell. When we give high resolution the solution takes more time and complexity to run. But very low resolution(i.e. High individual cell size) results in the path algorithm being unable to find the optimal solution. We experimented with values like 5,10,1(best) for resolution. A higher resolution means we can avoid the obstacle more precisely when safety is the only concern and not the time to run.
     - Define the position of the obstacle by marking all the points on the boundary of the obstacle. We also mark the endpoints of the grid as obstacles.
     - The algorithm uses True to indicate the cell occupied by an obstacle and False indicates free space. The algorithm marks all the cells that are in the robot radius as occupied.
     - ASSIGNING COST: For points around obstacles within the robot's radius, they are considered as obstacles and assigned an infinitely high cost. For all other points, an equal cost is assigned so that the algorithm can search across all possible paths and choose the path with the least cost.
   - Algorithm
     - Start with the initial node (start_node) and compute its cost (0).
     - Iteratively expand the least-cost node from the open set:
       - Generate its neighbours based on the motion model. Moving in 8 directions allows better path planning to find safer routes in very restrictive environments.
       - Verify neighbours are within bounds, not obstacles, and not already evaluated.
       - Add new valid neighbours to the open set with their respective costs.
     - If the goal node is reached, terminate the search.
     - The path is reconstructed using the parent_index attribute for each node.
   - Dijkstra's algorithm explores all possible paths, ensuring the safest path is found given the obstacle information. By assigning equal cost to all non-obstacle cells, it doesn't prioritise shorter paths over safer ones.

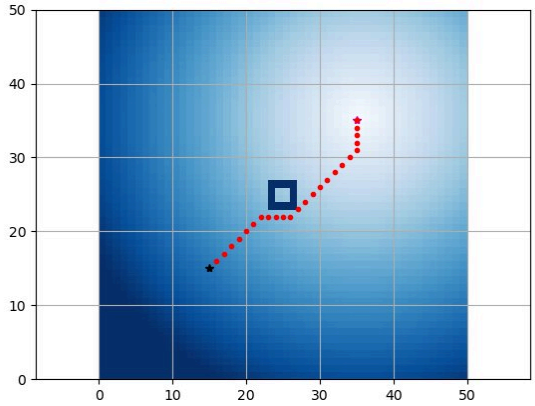
- **Disadvantage:** It is computationally intensive for large grids making it impractical in large-scale applications. The model assumes a static environment which may not be the case in real-world applications.

2. Minimum Time/Distance: A star
   A* is also an exact method representation. It finds the shortest path and uses a heuristic to prioritize nodes, reducing the search space compared to Dijkstra's algorithm.

   - Initialization:
     - Define the position of the obstacle by marking all the points on the boundary of the obstacle. We also mark the endpoints of the grid as obstacles.
     - We used a grid resolution of 1 (grid cell size) as the optimal resolution.
     - Though this algorithm is made for the unsafe path, we gave a small robot radius to make sure the robot doesn't crash the obstacle and stays a little farther.
   - A Search*:
     a. Start with the initial node and compute its cost (0).
     b. Maintain open and closed (nodes which are already evaluated) sets for nodes.
        1. Generate neighbours based on the motion model. Moving in 8 directions allows better path planning to find safer routes in very restrictive environments.
        2. Verify neighbours are within bounds, not obstacles, and not already evaluated. So it's more unsafe than the Dijkstra algorithm.
        3. Add valid neighbours to the open set with their costs. If the new path to a neighbour is better, update the neighbour's cost and parent.
     c. Select the node with the lowest cost + heuristic (g+h) value from the open set. g: This is similar to the cost in the Dijkstra algorithm. It says how many units we moved from the start node. h: It gives an estimate of where the goal node is.
     d. Iteratively expand the node with the lowest cost (f = g + h).
        1. Heuristic Function: we use Euclidean distance to estimate the cost from the current node to the goal node. d = w * math.hypot(n1.x - n2.x, n1.y - n2.y) We used w = 1 for our algorithm. For future works, we can try w >1 i.e. weighted A star.
     e. Terminate when the goal node is reached. The path is reconstructed using the parent_index attribute for each node.
   - **Disadvantages:** Performance decreases for large or high-resolution grids. An inaccurate heuristic can degrade performance. Also the problem of algorithms not generalizing for dynamic environments.

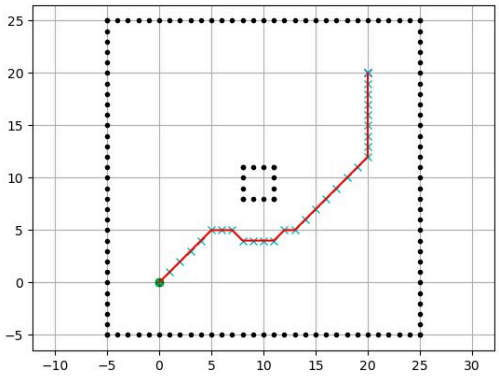
Other algorithms we experimented with:

1. Potential Field Planning
   The path given by this algorithm is [[0.0, 0.0, 0.0], [0.1, 0.1, 0.0], [0.2, 0.2, 0.0], [0.3, 0.3, 0.0], [0.4, 0.4, 0.0], [0.5, 0.5, 0.0], [0.6, 0.6, 0.0], [0.7, 0.7, 0.0], [0.8, 0.7, 0.0], [0.9, 0.7, 0.0], [1.0, 0.7, 0.0], [1.1, 0.7, 0.0], [1.2, 0.8, 0.0], [1.3, 0.9, 0.0], [1.4, 1.0, 0.0], [1.5, 1.1, 0.0], [1.6, 1.2, 0.0], [1.7, 1.3, 0.0], [1.8, 1.4, 0.0], [1.9, 1.5, 0.0], [2.0, 1.6, 0.0], [2.0, 1.7, 0.0], [2.0, 1.8, 0.0], [2.0, 1.9, 0.0], [2.0, 2.0, 0.0]. Visualization of this path is shown in the following image.



We tried this algorithm for minimum distance/time, but the distance of the path is longer than the path given by A*.

2. Greedy Best First Search
   The path given by this algorithm is [[0.0, 0.0, 0.0], [0.1, 0.1, 0.0], [0.2, 0.2, 0.0], [0.3, 0.3, 0.0], [0.4, 0.4, 0.0], [0.5, 0.5, 0.0], [0.6, 0.5, 0.0], [0.7, 0.5, 0.0], [0.8, 0.4, 0.0], [0.9, 0.4, 0.0], [1.0, 0.4, 0.0], [1.1, 0.4, 0.0], [1.2, 0.5, 0.0], [1.3, 0.5, 0.0], [1.4, 0.6, 0.0], [1.5, 0.7, 0.0], [1.6, 0.8, 0.0], [1.7, 0.9, 0.0], [1.8, 1.0, 0.0], [1.9, 1.1, 0.0], [2.0, 1.2, 0.0], [2.0, 1.3, 0.0], [2.0, 1.4, 0.0], [2.0, 1.5, 0.0], [2.0, 1.6, 0.0], [2.0, 1.7, 0.0], [2.0, 1.8, 0.0], [2.0, 1.9, 0.0], [2.0, 2.0, 0.0], [2.0, 2.0, 0.0]]
   Visualization of this path is shown in the following image.

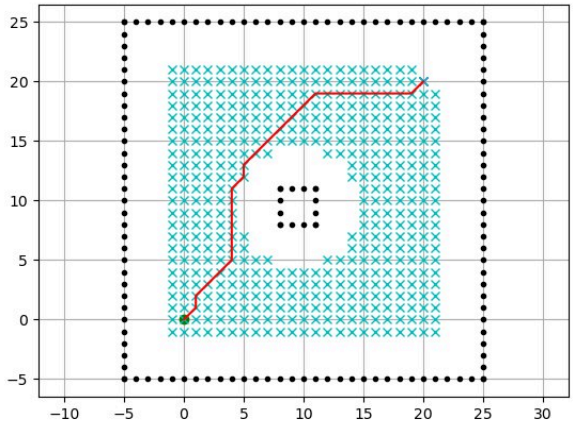


This path is similar to the one given by Dijkstra.

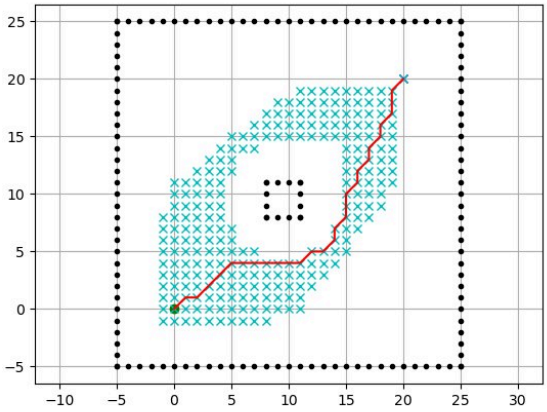
Visualization

In the plots, black dots represent walls and obstacles, blue dots represent points that are considered for the path, and red lines represent the final path provided by the planning algorithm.

1. Maximum Safety - Dijkstra



2. Minimum Time/Distance - A*



Implementation

We use Dijkstra and A* algorithm to plan the path for 2 tasks, adjust them to the scale of our environment and set the path as waypoints. While the robot is moving, it detects the april tags and adjusts its current state based on the april tag positions.

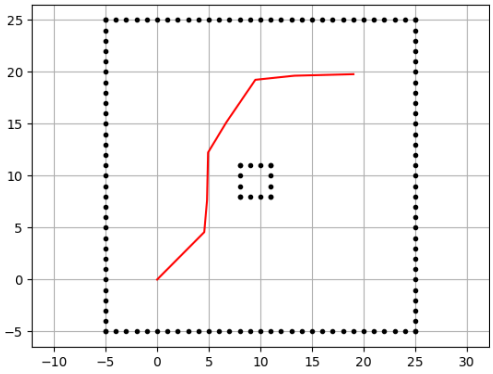
1. Maximum Safety - Dijkstra
    a. The waypoints planned by the Dijkstra algorithm is [[0.0, 0.0, 0.0], [0.1, 0.1, 0.0], [0.1, 0.2, 0.0], [0.2, 0.3, 0.0], [0.3, 0.4, 0.0], [0.4, 0.5, 0.0], [0.4, 0.6, 0.0], [0.4, 0.7, 0.0], [0.4, 0.8, 0.0], [0.4, 0.9, 0.0], [0.4, 1.0, 0.0], [0.4, 1.1, 0.0], [0.5, 1.2, 0.0], [0.5, 1.3, 0.0], [0.6, 1.4, 0.0], [0.7, 1.5, 0.0], [0.8, 1.6, 0.0], [0.9, 1.7, 0.0], [1.0, 1.8, 0.0], [1.1, 1.9, 0.0], [1.2, 1.9, 0.0], [1.3, 1.9, 0.0], [1.4, 1.9, 0.0], [1.5, 1.9, 0.0], [1.6, 1.9, 0.0], [1.7, 1.9, 0.0], [1.8, 1.9, 0.0], [1.9, 1.9, 0.0], [2.0, 2.0, 0.0]]
    b. If the next waypoint is too close to the previous one, the speed of the robot will be so low that the wheels don't rotate because of the friction of the ground. Therefore, we adjust the path given by the Dijkstra and give the following way points to the robot: [[0.0, 0.0, 0.0],[0.75,0.75,0.0],[0.75,1.95,0],[1.5,3,0],[3,3,0.0]]

c. The actual current state/path traversed by the robot: [(0.26527327, 0.26527327), (0.49649071, 0.49649071), (0.69170573, 0.69170573), (0.73270013, 1.15713737), (0.74095227, 1.5353374), (0.74791948, 1.85464865), (1.01465584, 2.29308536), (1.24616779, 2.6302877), (1.44163148, 2.91498494), (2.01322464, 2.97477026), (2.48392215, 2.98680499), (2.88132829, 2.99696582)] (We use the weighted mean of robot coordinate in the world frame obtained by April tag detection from hw2 and the current state. This adjustment accounts for the high errors in y coordinates obtained from April tag detection as done in HW3 using Q &R matrices.)

2. Minimum Time/Distance - A*
   a. The waypoints planned by the A* algorithm is [[0.0, 0.0, 0.0], [0.1, 0.1, 0.0], [0.2, 0.1, 0.0], [0.3, 0.2, 0.0], [0.4, 0.3, 0.0], [0.5, 0.4, 0.0], [0.6, 0.4, 0.0], [0.7, 0.4, 0.0], [0.8, 0.4, 0.0], [0.9, 0.4, 0.0], [1.0, 0.4, 0.0], [1.1, 0.4, 0.0], [1.2, 0.5, 0.0], [1.3, 0.5, 0.0], [1.4, 0.6, 0.0], [1.4, 0.7, 0.0], [1.5, 0.8, 0.0], [1.5, 0.9, 0.0], [1.5, 1.0, 0.0], [1.6, 1.1, 0.0], [1.6, 1.2, 0.0], [1.7, 1.3, 0.0], [1.7, 1.4, 0.0], [1.8, 1.5, 0.0], [1.8, 1.6, 0.0], [1.9, 1.7, 0.0], [1.9, 1.8, 0.0], [1.9, 1.9, 0.0], [2.0, 2.0, 0.0]]
   b. We adjust the path given by the A* and give the following way points to the robot: [[0.0, 0.0, 0.0],[1,1,0.0],[2.2,1,0],[4,3.5,0.0]]
   c. The actual current state/path traversed by the robot: [(0, 0), (0.36880698503783266, 0.3427828744032369), (0.67205941, 0.65471182), (0.92765468, 0.91838757), (1.40432395, 0.97480013), (1.783866, 0.98682061), (2.1060792122757364, 0.9817706753894921), (2.806681130223363, 1.8515640031506302), (3.37552132, 2.63463072), (3.85586525, 3.29642355)] (We use the weighted mean of robot coordinate in the world frame obtained by April tag detection from hw2 and the current state. This adjustment accounts for the high errors in y coordinates obtained from April tag detection as done in HW3 using Q &R matrices.)

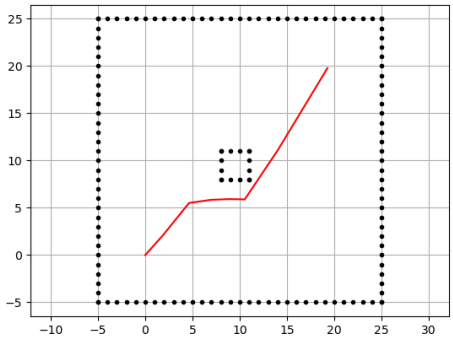
Results

1. Maximum Safety
   Video: https://youtu.be/6w7u2KB__i4
   The actual trace of the robot is plotted in the following graph



In this task, the robot keeps some distance away from the obstacle to maximize safety.

2. Minimum Time/Distance
   Video: https://youtu.be/taelVyL0__M
   The actual trace of the robot is plotted in the following graph



   In this task, the robot brushed past the obstacle.

References

We referenced this (https://github.com/AtsushiSakai/PythonRobotics/tree/master/PathPlanning) github repo for the path planning algorithm implementation.