CSE 276A - Introduction to Robotics

HW4

Drishti Megalmani A59001657
Sai Ashritha Kandiraju A59001630

In this homework, we aim to implement path planning for our robot in an environment with obstacles (defined by april tags).

METHODOLOGY

SAFEST PATH

Video Link: https://youtu.be/qfcVVSgj6gE

Objective - In the path planning algorithm for maximum safety, we define our objective for safety as "being as far away as possible from the obstacles". Thus in obtaining a maximal safe path to the goal, the robot will never come closer to the obstacles than what is required.

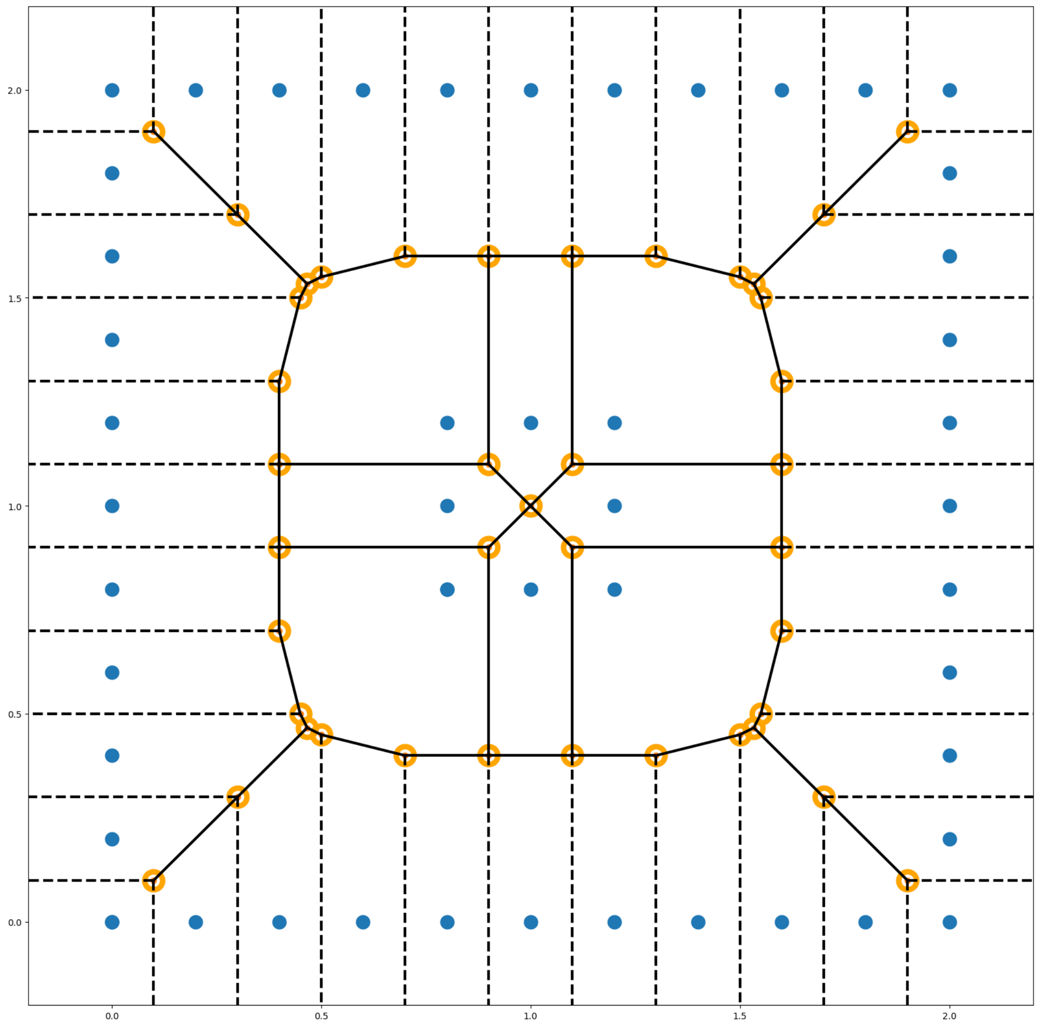
Representation - In our approach the environment (with obstacles) is considered to be a set of 2D points. We then represent this 2D map as a Voronoi graph.

Voronoi Diagram Creation

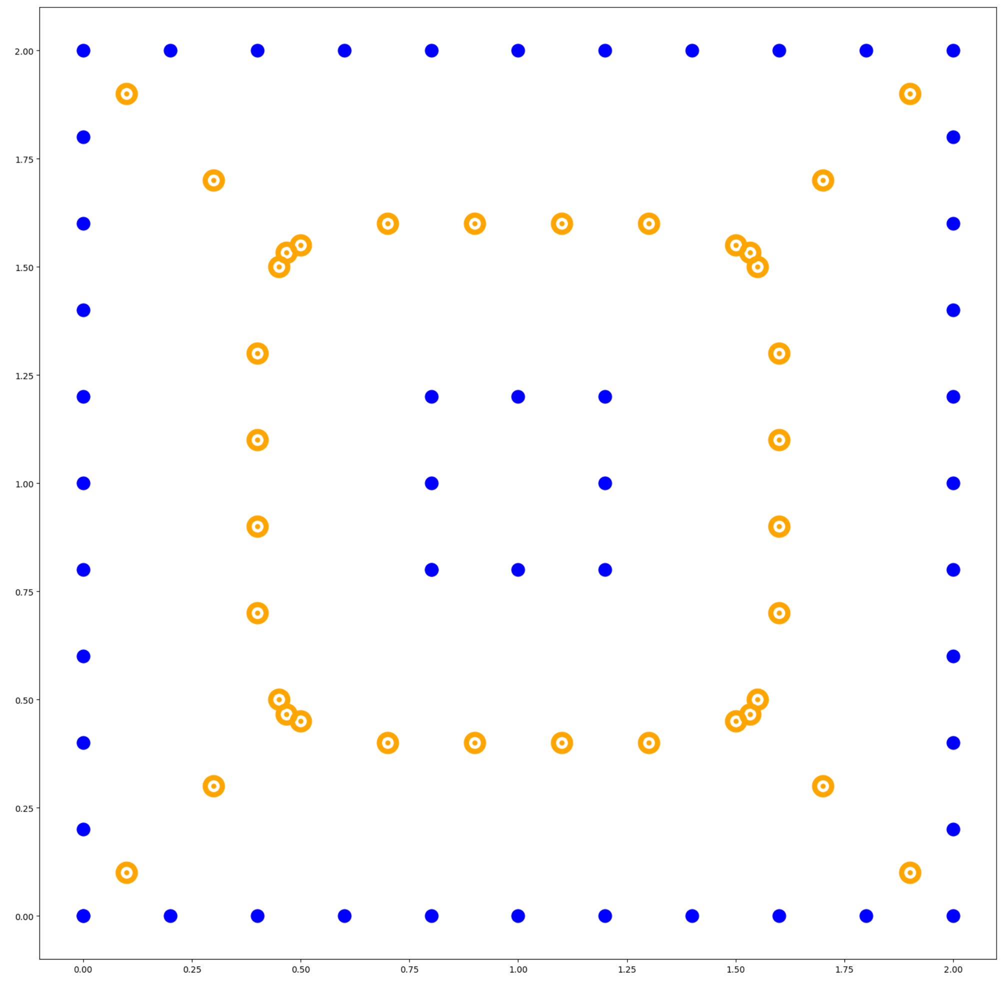
Voronoi diagram is the partitioning of a plane with n points into convex polygons such that each polygon contains exactly one generating point and every point in a given polygon is closer to its generating point than to any other. 2D-Voronoi diagrams have a special property that helps us with designing a path for robots given a set of obstacles. There will always be a perpendicular bisector lying between two neighboring points. Suppose all the points come from the edges of the obstacles, then the bisectors will only have two possible properties. If the two points are from the same obstacle edge, the bisector will definitely have at least one endpoint lying inside the obstacle. On the other hand, if the two points are from different obstacles, then the bisector will lie between these two obstacles, and that is a valid path we can consider. This is the ideology behind Voronoi diagrams. Thus, representing the world as a Voronoi graph will ensure that the robot is a maximum distance from all the obstacles.

- We initialized an external boundary to define the outer boundary of 8 landmarks with a contiguous set of points eventually representing a square of size 7ft x 7ft.
- To define the obstacle in the middle, we considered:
  - A single point representation where the size of the obstacle wouldn't be considered
  - A set of points defining the boundary of the square to incorporate the size of the obstacle as well.
- We used the *scipy* implementation for Voronoi diagrams to eventually get the list of vertices that can be considered for our traversal.

In diagram 1, we see the Voronoi diagram for our setup. The blue points represent the outer boundary (containing the landmarks) and the central obstacle. The orange points represent the vertices that are eligible candidates for our path. The lines represent the polygons. The solid lines are possible paths. An issue we noticed due to the representation of the solid obstacle as a boundary of sparse points, was that certain points of traversal were also present within the obstacle.



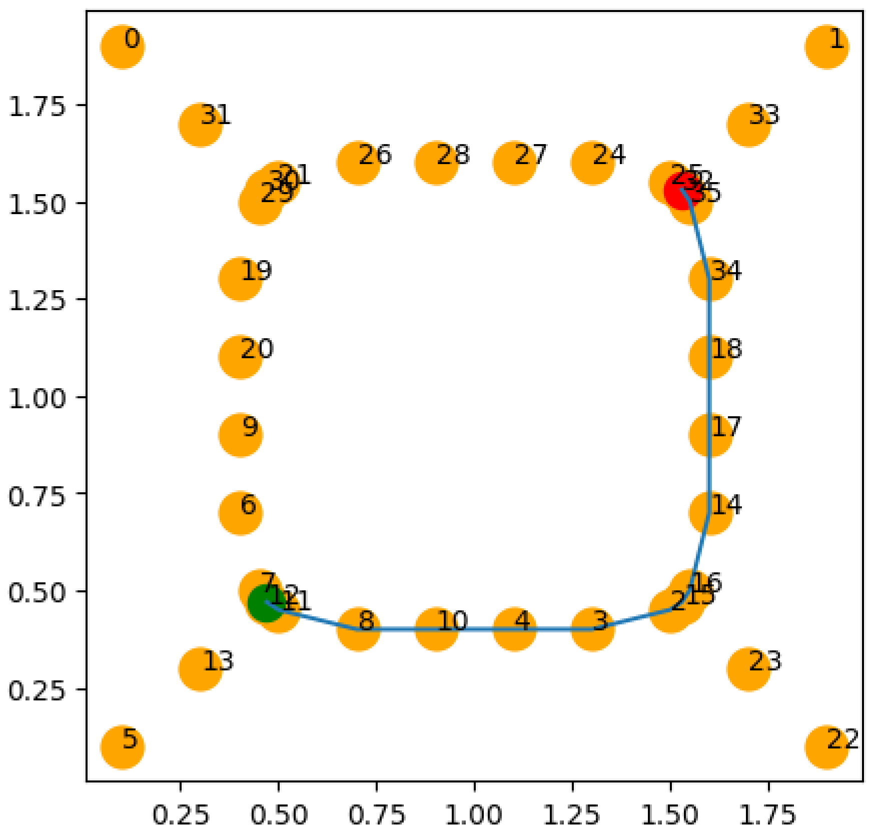
To eliminate this, we added another check to post-process the vertices we get. This check was to eliminate all the vertices within the obstacles. The final plot with obstacles (blue) and candidate vertices for path (orange) is given below.



Configuration Space - The orange vertices in the above diagram can be considered to be the configuration space for the robot. To account for the build of the robot (which isn't point size), we found critical points in the Voronoi diagram. These are the points where the Voronoi path has a local minima. At these points, we manually checked to see if the robot's diameter would fit through the available space at the critical point. Since we assumed a fixed orientation of the robot throughout its motion, we did not worry about all the other rotations of the robot.

BFS Algorithm

- We extracted the Voronoi vertices
- We mapped each vertex to a numbered node to make it easier to represent it as a graph for BFS.
- We generated a graph from these numbered nodes by representing all the edges possible as an adjacency list.
- We then ran BFS to determine the shortest path from start node 12 to destination node 32. The green node is the start node and the red node is the destination node in the figure.
- We set the waypoints to the path we received from the BFS algorithm. The figure below shows the path followed by the robot.



```
[[0.5, 0.45, 0.0],[0.7, 0.4, 0.0],[0.9, 0.4, 0.0],
[1.1, 0.4, 0.0],[1.3, 0.4, 0.0],[1.5, 0.45, 0.0],
[1.53, 0.47, 0.0],[1.55, 0.5, 0.0],[1.6, 0.7, 0.0]
[1.6, 0.9, 0.0], [1.6, 1.1, 0.0], [1.6, 1.3, 0.0],
[1.55, 1.5, 0.0], [1.53, 1.53, 0.0]]
```

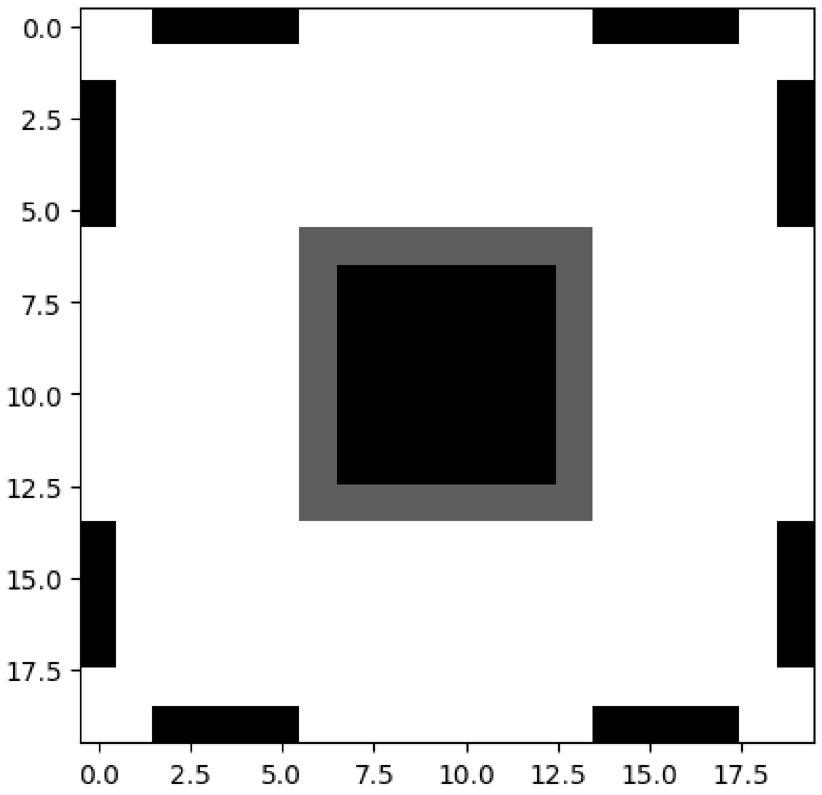
Why is it the safest?

In step 1, where we construct the Voronoi diagram for the given set of obstacle points, we are reducing the sample space of the possible waypoints that the robot can take. The resulting Voronoi Diagram gives the set of points where the distance to the two nearest objects/obstacles is the same. Hence, the output Voronoi vertices that we get are at the most optimal distance in terms of safety from all the obstacles around it. Picking a path from source to destination from these sets of resulting points via BFS thereby gives the safest path.

SHORTEST PATH
Video Link: https://youtu.be/CJA4Lq_654g

Objective - In the path planning algorithm for minimum distance, we define our objective for minimal distance as "moving from the source to the destination with the shortest distance/path". We are considering Euclidean distance between two waypoints to do so.
Representation - In this algorithm, we aren't reducing the sample space to the optimal set of vertices but considering the entire environment space and discretizing the environment into a grid space. Occupancy grid maps discretize a space into squares of arbitrary resolution and assign each square either a binary or probabilistic value of being full or empty.

We are creating a grid with each cell of size 0.1m x 0.1m. Therefore creating a grid of size 20x20 owing to the 2mx2m environment space. The black regions represent the obstacles/ april tags and the white spaces represent the empty space where the robot can possibly travel in. In this method, we also added a safety grid around the obstacle to avoid collisions since the robot isn't a point and a solid entity. This region is represented by a dark grey region. The lower left corner is considered as our origin.



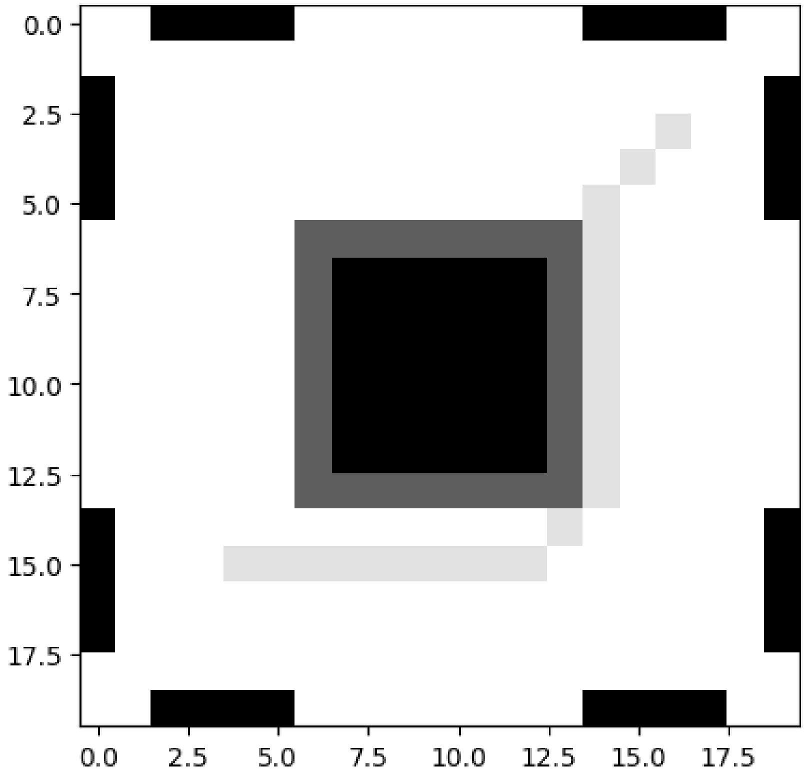
A* Algorithm
A* is a path-finding algorithm that extends Dijkstra's algorithm by adding heuristics to stop certain unnecessary nodes from being searched. This is done by weighting the cost values of each node distance by their euclidean distance from the desired endpoint. Therefore, only paths

that are headed generally in the correct direction will be evaluated. A* selects the path that minimize the cost function based on this formula:

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

where g(n) is the cost of the edge and h(n) is the value of the heuristic. Two kinds of very common heuristics are the Manhattan distance and the Euclidean distance.

In our implementation we represented the environment as a grid matrix as shown above and used the *networkx* library to convert the grid into a graph and eventually used the *astar_path()* function with the default parameters and our graph, start and end nodes to get the path shown by the grid image below:



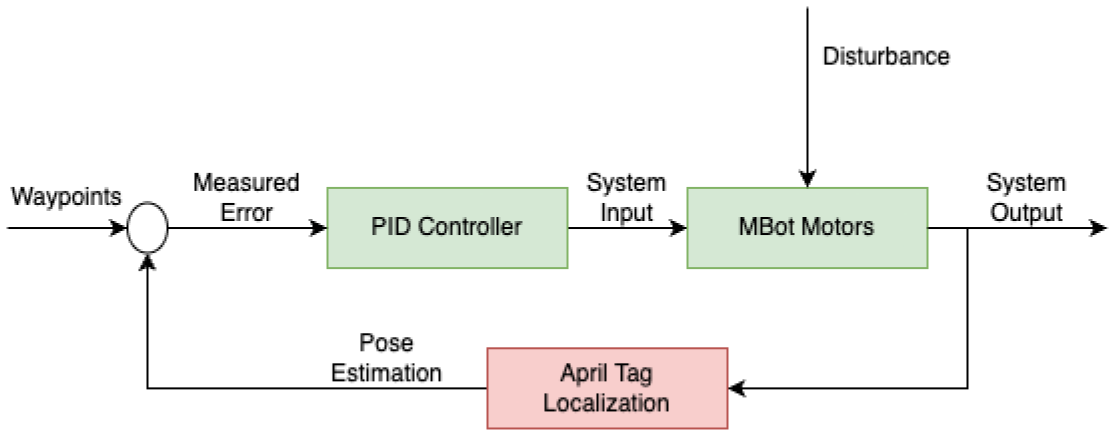
The waypoints of the path are given by:

```
[[0.4, 0.4, 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, 0.4, 0.0],
[1.3, 0.5, 0.0],[1.4, 0.6, 0.0],[1.4, 0.7, 0.0],
[1.4, 0.8, 0.0],[1.4, 0.9, 0.0],[1.4, 1.0, 0.0],
[1.4, 1.1, 0.0],[1.4, 1.2, 0.0],[1.4, 1.3, 0.0],
[1.4, 1.4, 0.0],[1.4, 1.5, 0.0],[1.6, 1.6, 0.0]]
```

Why is it the shortest?
We believe that this algorithm provides the shortest since we are considering all the possible locations (and not just the safest locations) in the empty space in the environment and running the shortest path search algorithm A* on it.

IMPLEMENTATION DETAILS

Architecture



We used the same framework as in the HW2 solution with some changes to make it compatible with the april detection node (The HW2 solution provided didn't really work without modifications due to the discrepancy in publish and subscribe messages of the tf.transform).
In the planner node (hw4_solution), we use a listener to obtain the measurement from april tags and transform this to get the current pose of the robot in the world. The planner node then publishes a twist message to the mpi control node which moves the motors.

Handling Same Tag Ambiguity

Since the map has the same tags placed at multiple locations, choosing the right pose matrix for the transformations and localization is essential.
To handle this ambiguity, we followed these steps:

- The *`current_state`* of the robot (from the open loop update step or last known current_state after localization) corresponds to *`wTr.`*
- The measurement from the april tag detection will give us `cTa`.
- We already know the transformation matrix `rTc`.
- We perform the following transformations:

```
rTa = rTc x cTa
wTa = wTr x rTa
```

- `wTa` april tag in the world coordinates which is nothing but our values in the april tag pose matrices.
- For each tag id, we maintain a list of possible pose matrices in the map based on each occurrence. We find the error in terms of the euclidean distance between the wTa's

translation coordinates and the coordinates of each tag in the april_tag pose matrices. The tag with least error, is our required tag. Here, we are basically finding the error between our estimate of the april tag in the world with the actual pose of that april tag.

CHALLENGES

1. One of the april tags gave us a lot of variance in terms of localization and we faced a lot of issues with that.
2. We had to re-tune the PID, calibration and threshold error due to inconsistent and jerky motion.

CONTRIBUTION

Both of us contributed equally on all sections of the homework (from calibration, design thinking, implementation, testing and report).