American University of Sharjah

College of Engineering

Department of Computer Science and Engineering

Spring 2024

CMP 49410 - Intelligent Autonomous Robotics

Project 2

Ahmad Alsaleh - @00093749

Ahmed Alabd Aljabar - @00092885

Omar Ibrahim - @00093225

Yousef Irshaid - @00093447

Submitted To: Dr. Michel Pasquier

Submission Date: 14th May 2024

Project 2 Answers

Planning (Extra):

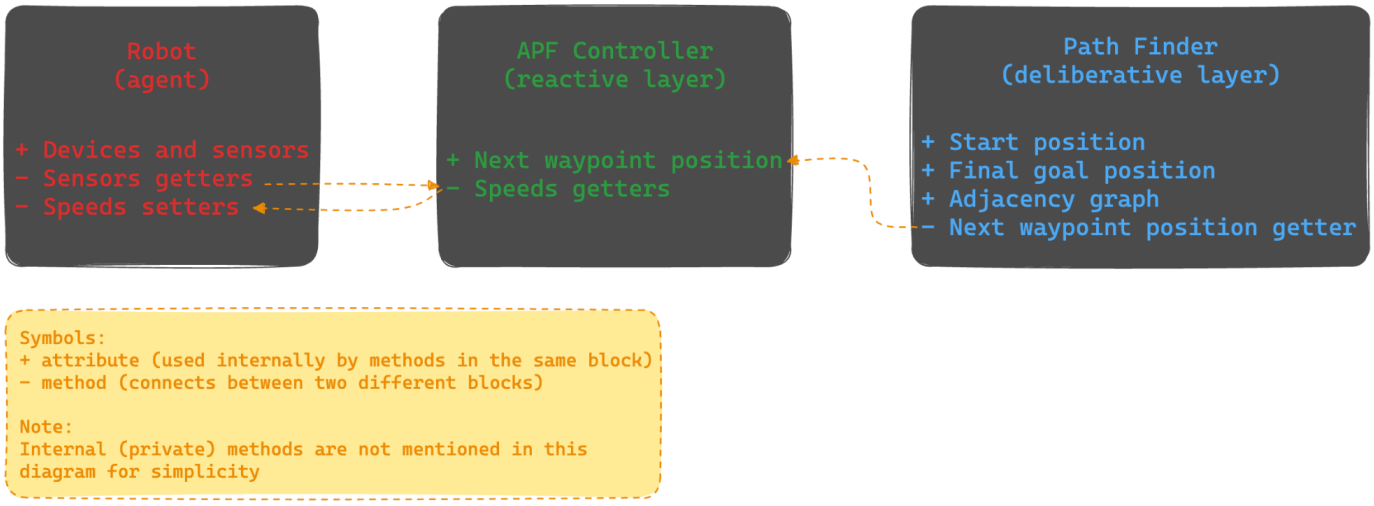
After reading the project requirements, we started by identifying the different layers we needed to implement. Based on that, we created the below graph.[1] Later when we start coding, each block in the graph will be roughly mapped to a class in the code.

At the end of the day, our goal is to compute the speeds of the robot motors. Below, we explain how each block interacts with the other blocks to achieve this goal.

The first block is the `Robot` block, which will contain all the devices and sensors of the robot agent. It will also provide getters for all sensors to give other blocks access to the readings of these sensors.

The second block is the `Path Finder` block, which will hold the start and goal positions as well as the map of the environment. It will run a path-finding algorithm like A* and produce the next waypoint of the generated path.

The third block is the `APF Controller` block, which will receive the next waypoint from the `Path Finder` block and the sensor readings from the `Robot` block. It will use this information to compute the speeds of the robot wheels. Finally, the computed speeds will be given to the `Robot` block to move the robot agent in the correct direction.



[1] Graph generated using excalidraw.com

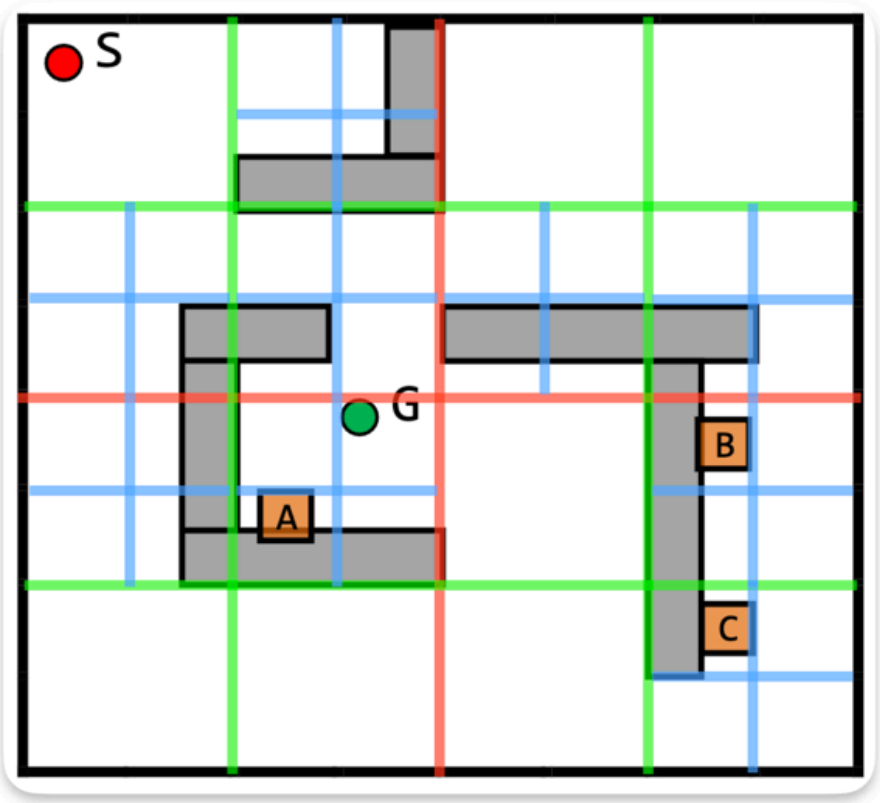
Question 1

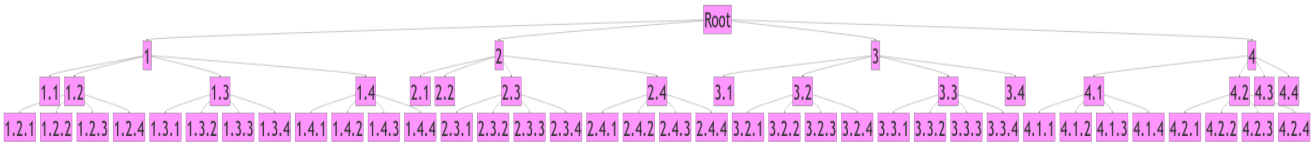
a)

- One problem with using a uniform grid for path planning is that all cells have the same size, which can lead to a lot of unnecessary cells being created. This can lead to a lot of wasted memory and computation time. This happens when the resolution of the grid is too high, i.e. smaller cell sizes.
- On the contrary, when the resolution is too low, i.e. larger cell sizes, the path planning algorithm may not be able to find a path around the obstacle, since the whole cell is considered as an obstacle, where in reality the robot might be able to pass through the empty part of the cell.
- Also, if the resolution is too low, the path may not be smooth and may have sharp turns.
- The pros of increasing the grid resolution are:
    - More accurate path planning.
    - Smoother paths.
    - Better environment representation in real-world scenarios.
    - Better obstacle avoidance.
- The cons of increasing the grid resolution are:
    - More memory usage.
    - More computation time.

b)

For the quadtree, we are assuming the numbering of the quads goes clockwise, starting from the top left. That is, the top left quad is quad 1, the top right is quad 2, the bottom right is quad 3, and the bottom left is quad 4. This numbering scheme goes recursively. So, for example, station B in the below graph is in box 3.2.1. That is, it is in the 3rd quad of level 1, 2nd quad of level 2, and the 1st quad of level 3. Note: sorry, we could not make the font in the tree structure larger as the tree is too wide.[2]

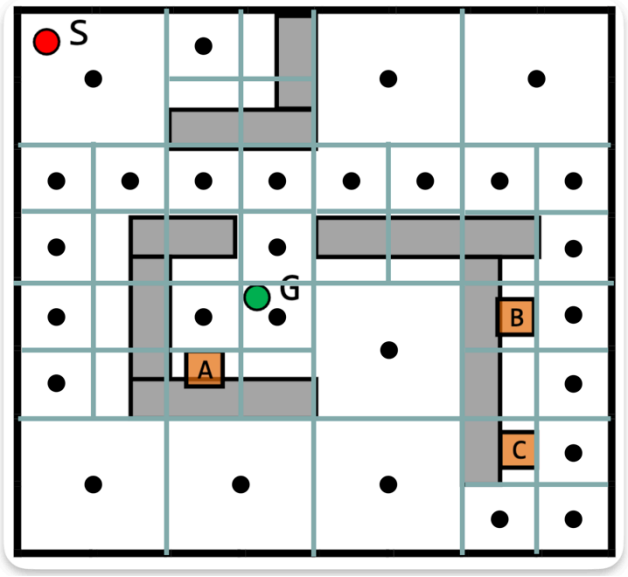




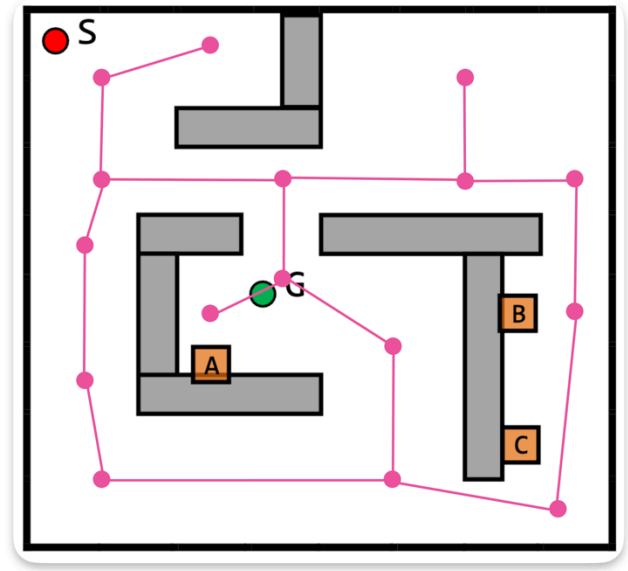
[2] The tree structure was generated using mermaid.live. Use the script available in the mmd file submitted with the zip file for a better preview resolution.

c)

Below are the generated waypoints.[3]



Below is the optimized topological map of the connected waypoints. The number of waypoints was reduced from 28 to only 16 after optimization.

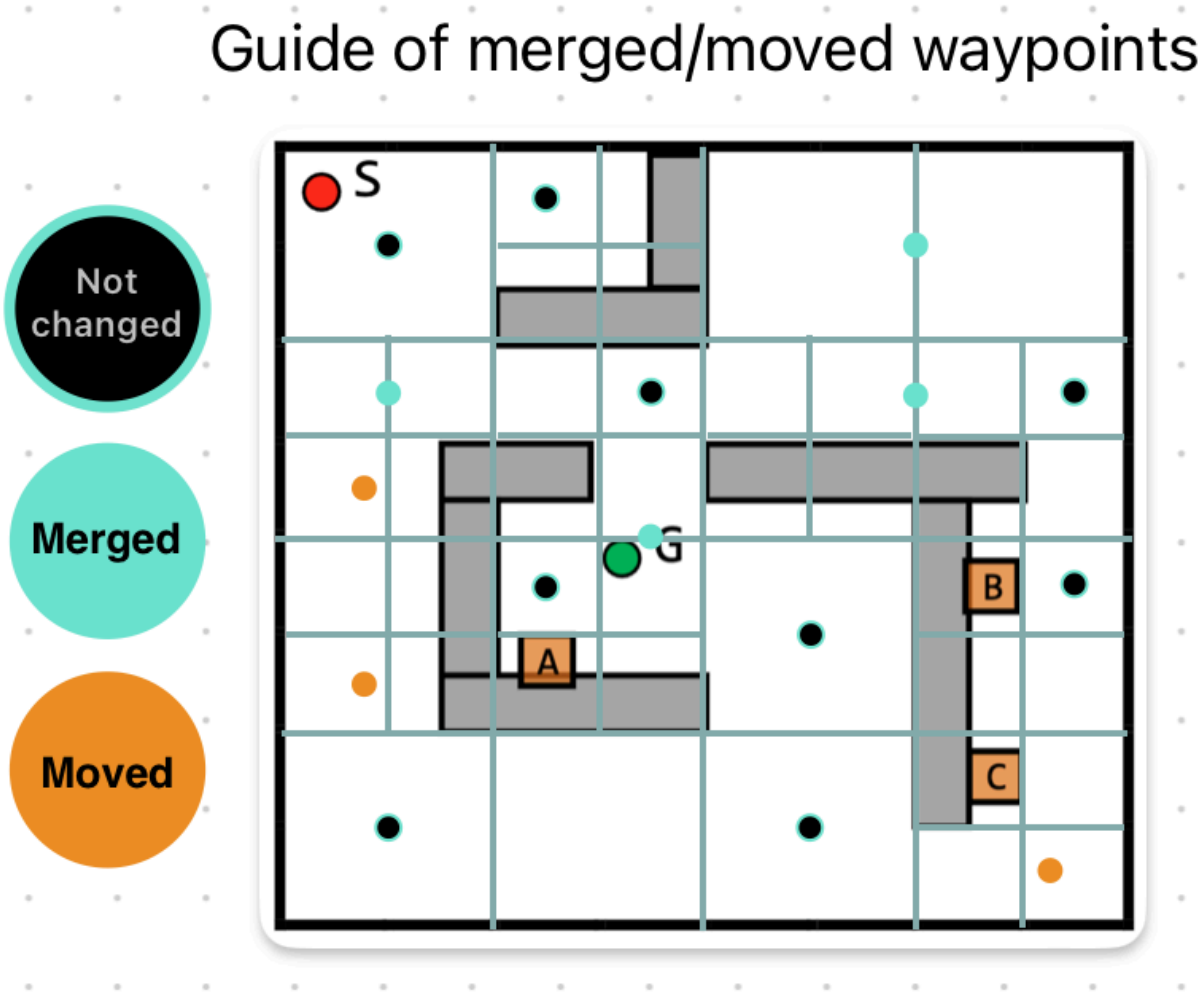


[3] Terminology: in this report and in the code, we use the term "waypoint" to refer to nodes. They mean the same thing.

Below is a color guide of the waypoints that were moved, merged, or unchanged.

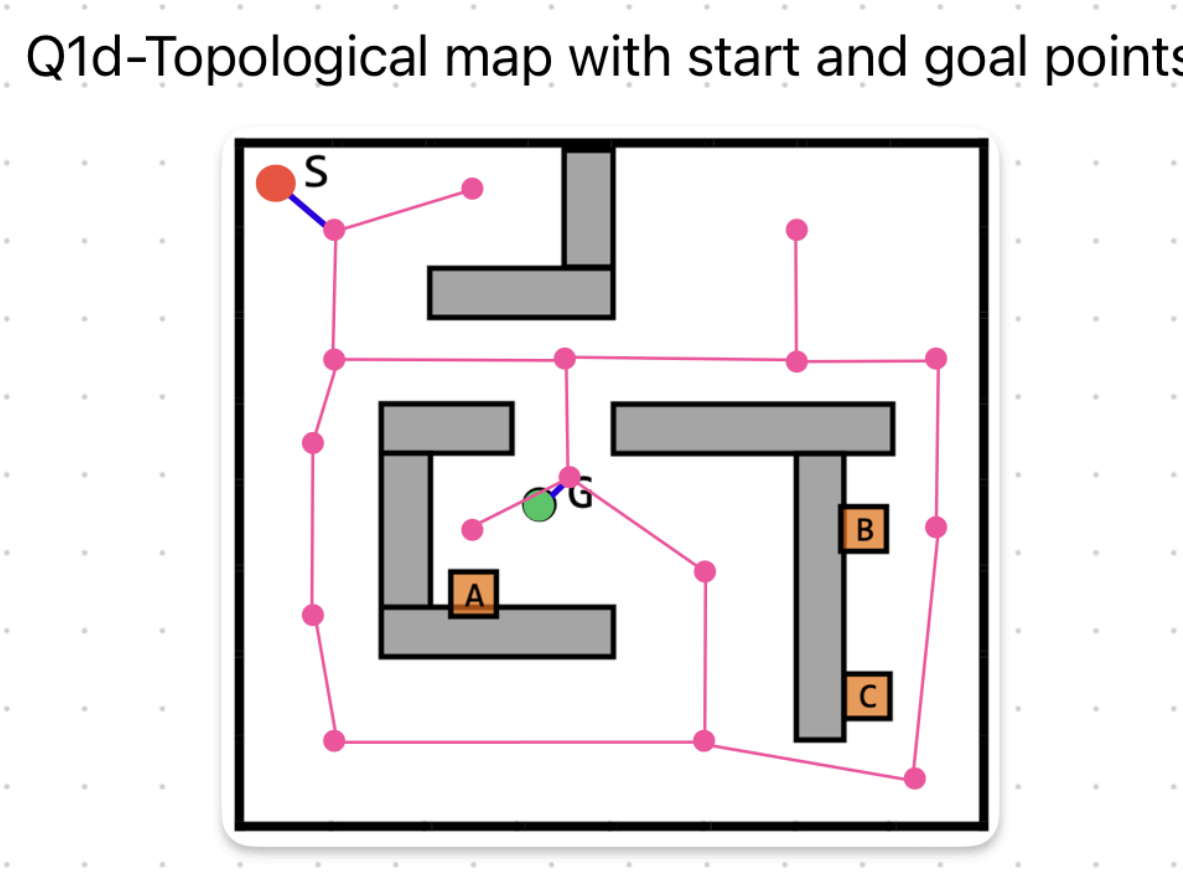
We followed the same philosophy to merge *any* two waypoints: waypoints are merged as long as the merging does not compromise the robot's ability to reach all areas. For example, the cyan dot that's slightly below the 'S' waypoint was produced from merging two waypoints since we didn't compromise the robot's ability to reach any other waypoint on the map.

We slightly moved some waypoints to allow a smoother turning of the robot and to make the robot evenly spaced from obstacles as much as possible. For example, the two orange dots on the left side were too close to the left wall, so we decided to slightly move them to the right.



d)

The below map takes the Start (S) and Goal (G) waypoints into consideration. Only two new edges are added to the graph (shown in blue), one for the S node and one for the G node. Everything else on the map is left as is.



The graph is implemented as a hard-coded dictionary where the keys-value pairs represent the nodes and the corresponding list of neighbors, respectively.

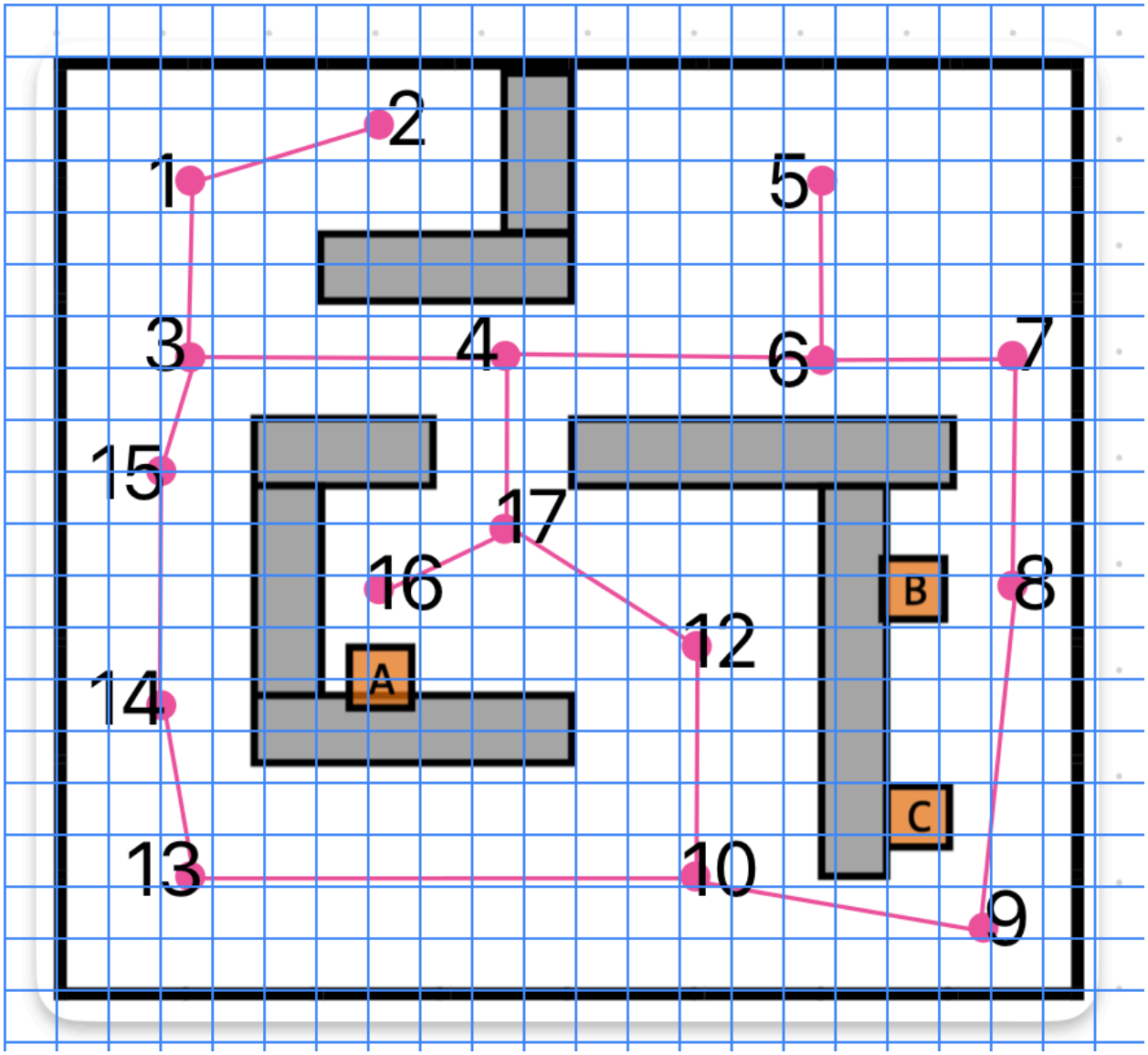
```
p1 = Waypoint(0.17, 0.97, "p1")
p2 = Waypoint(0.37, 1.05, "p2")
p3 = Waypoint(0.17, 0.78, "p3")
p4 = Waypoint(0.51, 0.78, "p4")
p5 = Waypoint(0.85, 0.97, "p5")
p6 = Waypoint(0.85, 0.78, "p6")
p7 = Waypoint(1.05, 0.78, "p7")
p8 = Waypoint(1.05, 0.49, "p8")
p9 = Waypoint(1.02, 0.07, "p9")
p10 = Waypoint(0.68, 0.15, "p10")
p12 = Waypoint(0.71, 0.41, "p12")
p13 = Waypoint(0.14, 0.15, "p13")
p14 = Waypoint(0.10, 0.34, "p14")
p15 = Waypoint(0.10, 0.63, "p15")
p16 = Waypoint(0.34, 0.49, "p16")
p17 = Waypoint(0.51, 0.60, "p17")

waypoints = {
    p1: [p2, p3],
    p2: [p1],
    p3: [p1, p4, p15],
    p4: [p3, p6, p17],
    p5: [p6],
    p6: [p4, p5, p7],
    p7: [p6, p8],
    p8: [p7, p9],
    p9: [p8, p10],
```

```
    p10: [p9, p12, p13],
    p12: [p10, p17],
    p13: [p10, p14],
    p14: [p13, p15],
    p15: [p3, p14],
    p16: [p17],
    p17: [p4, p12, p16],
}
```

Note: for ease of use and for organizational purposes, the above dictionary is stored in a separate file called `constants.py`. The dictionary could be obtained using the utility function `constants.get_waypoints()`.

The following map with a grid was used to *estimate* the (x, y) position of each node.[4] Note that the numbers were chosen arbitrarily.



[4] The grid was generated by yomotherboard.com/add-grid-to-image

The `Waypoint` data-class is an alias of the `Point` data-class, which stores the `(x, y)` coordinates of each node (see below).

Note that the `Point` data-class is made frozen to make the object immutable, hence hashable, allowing us to use it as a dictionary key, just like a normal tuple

```python
@dataclass(frozen=True)
class Point:
    x: float
    y: float
    name: str | None = None

    def __repr__(self) -> str:
        name = self.name if self.name is not None else "POINT"
        return f"{name}({self.x:.3f}, {self.y:.3f})"

    def to_numpy(self) -> np.ndarray:
        return np.array([self.x, self.y])

# type aliases
Waypoint = Point
Neighbors = List[Waypoint]
```

The `Graph` class takes an `adjacency_graph` (the `waypoints` dictionary for our case, see the previous code) and dynamically attaches the `start` and `goal` nodes to the closest nodes. The `Graph` class will also compute the cost and heuristic for each waypoint in the graph. Note that the `Graph` constructor accepts `start` and `goal` as arguments, allowing to easily change the start and goal waypoints.

```python
class Graph:
    def __init__(
        self,
        adjacency_graph: Dict[Waypoint, Neighbors],
        start: Waypoint,
        goal: Waypoint,
        cost_function: Callable[[Waypoint, Waypoint], float],
        heuristic_function: Callable[[Waypoint, Waypoint], float],
    ) -> None:
        self.__start = start
        self.__goal = goal
        self.__heuristic_function = heuristic_function

        # if the adjacency graph is not provided, create a simple
        # graph with the start and goal points connected to each other
        if not adjacency_graph:
            self.__adjacency_graph = {
                self.__start: [(self.__goal, cost_function(self.__goal, self.__start))],
                self.__goal: [(self.__start, cost_function(self.__start, self.__goal))],
            }
            return

        closest_to_start = (None, float("inf"))
        closest_to_goal = (None, float("inf"))
        self.__adjacency_graph: Dict[Waypoint, NeighborsWithCosts] = dict()
        for waypoint, neighbors in adjacency_graph.items():
            # finding the closest waypoint to the start
            distance = euclidean_distance(waypoint, self.__start)
            if distance < closest_to_start[1]:
                closest_to_start = (waypoint, distance)

            # finding the closest waypoint to the goal
            distance = euclidean_distance(waypoint, self.__goal)
            if distance < closest_to_goal[1]:
                closest_to_goal = (waypoint, distance)

            # appending the cost to each each neighbor
            # i.e.: {waypoint_1: [neighbor_1, neighbor_2, ...], ...} becomes
            # {waypoint_1: [(neighbor_1, cost_1), (neighbor_2, cost_2), ...], ...}
            # Note how neighbor_i was replaced by the tuple (neighbor_i, cost_i)
            self.__adjacency_graph[waypoint] = [
                (neighbor, cost_function(neighbor, waypoint)) for neighbor in neighbors
            ]

        # connecting the start to the rest of the graph
        closest_to_start = closest_to_start[0]
        self.__adjacency_graph[self.__start] = [
```

```
            (closest_to_start, cost_function(closest_to_start, self.__start))
        ]
        self.__adjacency_graph[closest_to_start].append(
            (self.__start, cost_function(self.__start, closest_to_start))
        )

        # connecting the goal to the rest of the graph
        closest_to_goal = closest_to_goal[0]
        self.__adjacency_graph[self.__goal] = [
            (closest_to_goal, cost_function(closest_to_goal, self.__goal))
        ]
        self.__adjacency_graph[closest_to_goal].append(
            (self.__goal, cost_function(self.__goal, closest_to_goal))
        )
```

e)
The path finding with A* algorithm is shown below[5]:

```
def a_star(graph: Graph) -> Path:
    """
    Returns the path found by A-Star.
    The heuristic and cost functions are provided by the `graph` object.
    """
    start = graph.get_start()
    goal = graph.get_goal()
    open_list = set([start])
    closed_list = set([])
    g = {start: 0}
    parents = {start: start}
    while len(open_list) > 0:
        n = None
        for v in open_list:
            if n == None or g[v] + graph.get_heuristic(v, goal) < g[
                n
            ] + graph.get_heuristic(n, goal):
                n = v
        if n == None:
            raise PathDoesNotExist
        if n == goal:
            reconstruction_path = []
            while parents[n] != n:
                reconstruction_path.append(n)
                n = parents[n]
            reconstruction_path.append(start)
            reconstruction_path.reverse()
            return Path(reconstruction_path)
        for m, weight in graph.get_neighbors(n):
            if m not in open_list and m not in closed_list:
                open_list.add(m)
                parents[m] = n
                g[m] = g[n] + weight
            else:
```

[5] The code is adapted from here with a few modifications to suit our needs

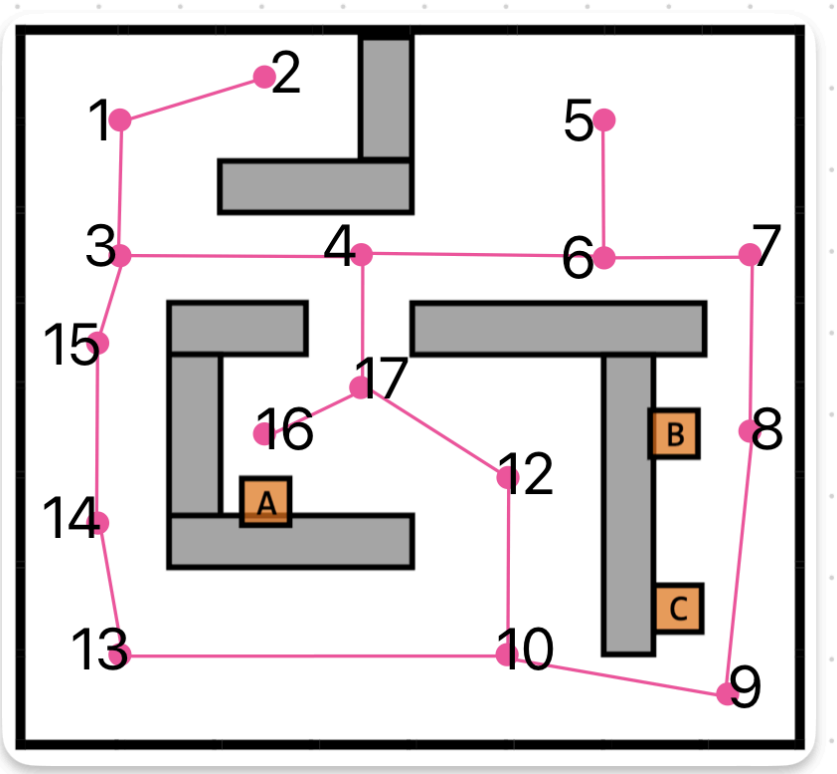
```
            if g[m] > g[n] + weight:
                g[m] = g[n] + weight
                parents[m] = n
                if m in closed_list:
                    closed_list.remove(m)
                    open_list.add(m)
    open_list.remove(n)
    closed_list.add(n)
raise PathDoesNotExist
```

Note that `a_star()` returns a `Path` object, which is nothing but a `list` object with the `__repr__` function overridden for pretty printing of the generated path.

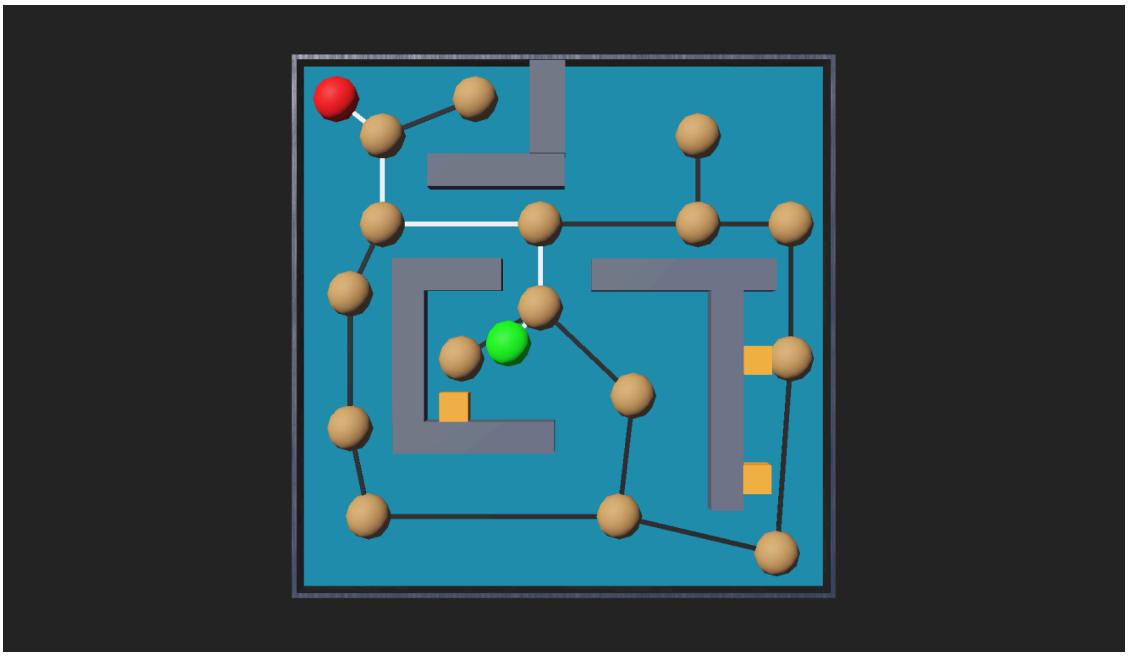
```
class Path(List[Waypoint]):
    def __repr__(self) -> str:
        return " --> ".join(map(str, self))
```

Test cases:

Please refer to the following picture for the number of each waypoint. Note that the numbers were chosen arbitrarily.



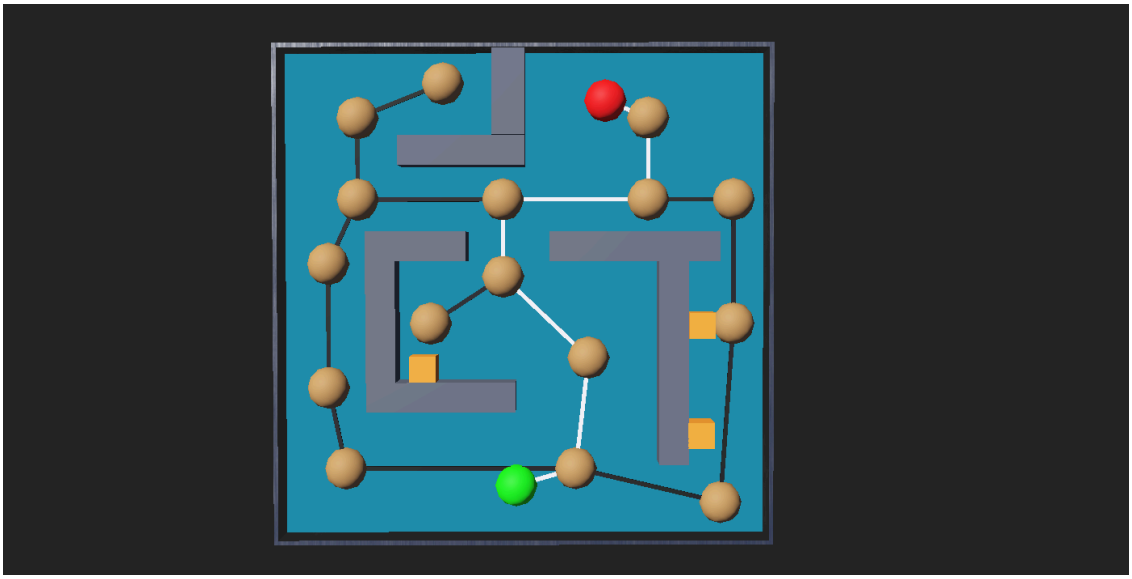
Test 1:



Shortest path:

```
start(0.070, 1.050) --> p1(0.170, 0.970) --> p3(0.170, 0.780) --> p4(0.510,
0.780) --> p17(0.510, 0.600) --> goal(0.441, 0.523)
```

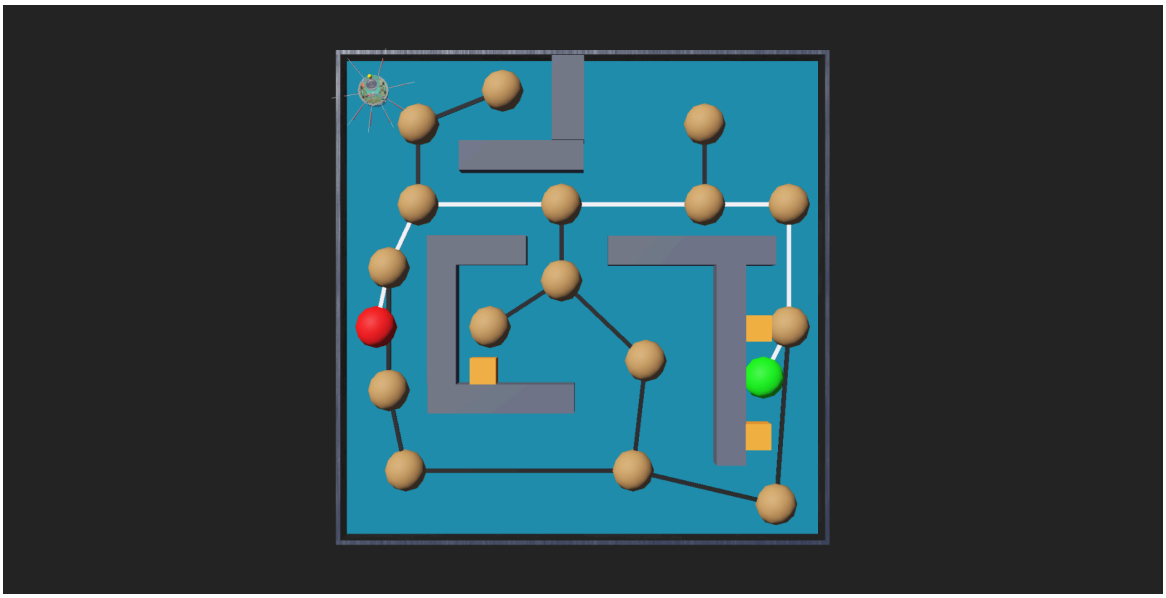
Test 2:



Shortest path:

```
start(0.750, 1.010) --> p5(0.850, 0.970) --> p6(0.850, 0.780) --> p4(0.510,
0.780) --> p17(0.510, 0.600) --> p12(0.710, 0.410) --> p10(0.680, 0.150) -->
goal(0.540, 0.110)
```

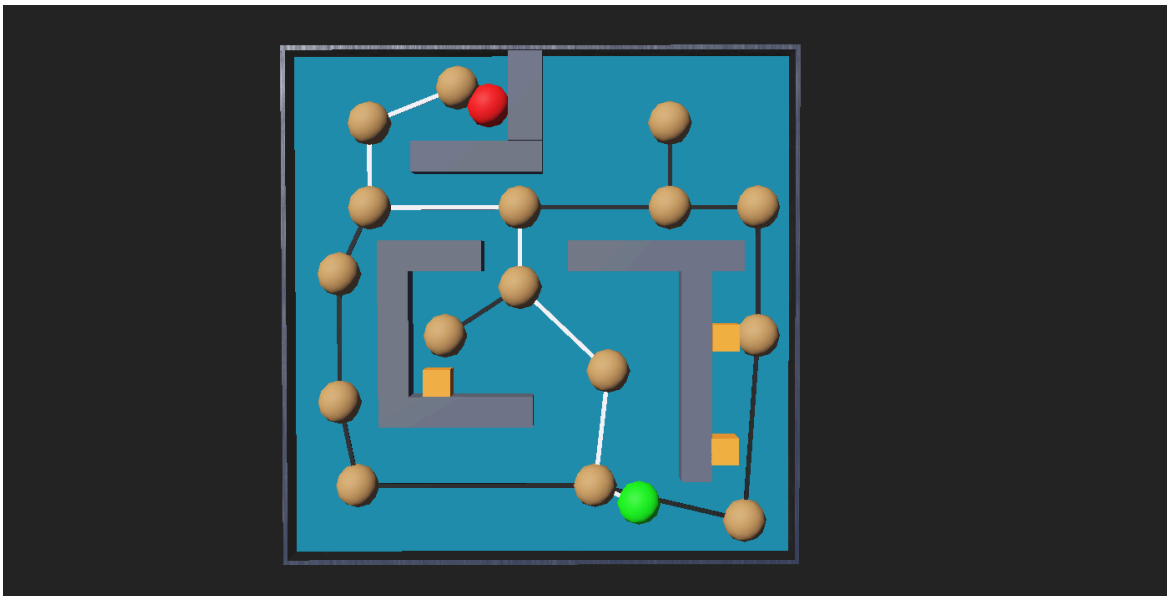
Test 3:



Shortest Path:

```
start(0.070, 0.490) --> p15(0.100, 0.630) --> p3(0.170, 0.780) --> p4(0.510,
0.780) --> p6(0.850, 0.780) --> p7(1.050, 0.780) --> p8(1.050, 0.490) -->
goal(0.990, 0.370)
```

Test 4:



Shortest path:

```
start(0.440, 1.010) --> p2(0.370, 1.050) --> p1(0.170, 0.970) --> p3(0.170,
0.780) --> p4(0.510, 0.780) --> p17(0.510, 0.600) --> p12(0.710, 0.410) -->
p10(0.680, 0.150) --> goal(0.780, 0.110)
```

f) The `DeliberativeLayer` class stores the overall planned path in `self.__path`. The `get_next_waypoint()` function returns the next waypoint in the path, which will be fed to the reactive controller. Note that `get_next_waypoint()` uses an `iterator` object to return a single waypoint. Doing so made the code much simpler to write and maintain as all we need to do is call `next()` to get the next waypoint.

```
class DeliberativeLayer:
    def __init__(self, graph: Graph) -> None:
        self.__path = a_star(graph)
        self.__path_iterator = iter(self.__path)

    def get_path(self) -> Path:
        return self.__path

    def get_next_waypoint(self) -> Waypoint:
        try:
            return next(self.__path_iterator)
        except StopIteration:
            raise PathTraversalCompleted("The path has been traversed.")
```

Below is the output of the `get_next_waypoint()` function for the four test cases. Note that two waypoints are given at every step.

- **Test 1:**

```
start(0.070, 1.050) reached. Going to p1(0.170, 0.970).
p1(0.170, 0.970) reached. Going to p3(0.170, 0.780).
p3(0.170, 0.780) reached. Going to p4(0.510, 0.780).
p4(0.510, 0.780) reached. Going to p17(0.510, 0.600).
p17(0.510, 0.600) reached. Going to p16(0.340, 0.490).
p16(0.340, 0.490) reached. Going to goal(0.440, 0.520).
goal(0.440, 0.520) reached. Final goal reached!!
```

- **Test 2:**

```
start(0.750, 1.010) reached. Going to p5(0.850, 0.970).
p5(0.850, 0.970) reached. Going to p6(0.850, 0.780).
p6(0.850, 0.780) reached. Going to p4(0.510, 0.780).
p4(0.510, 0.780) reached. Going to p17(0.510, 0.600).
p17(0.510, 0.600) reached. Going to p12(0.710, 0.410).
p12(0.710, 0.410) reached. Going to p10(0.680, 0.150).
p10(0.680, 0.150) reached. Going to goal(0.540, 0.110).
goal(0.540, 0.110) reached. Final goal reached!!
```

- **Test 3:**

```
start(0.070, 0.490) reached. Going to p15(0.100, 0.630).
p15(0.100, 0.630) reached. Going to p3(0.170, 0.780).
p3(0.170, 0.780) reached. Going to p4(0.510, 0.780).
p4(0.510, 0.780) reached. Going to p6(0.850, 0.780).
p6(0.850, 0.780) reached. Going to p7(1.050, 0.780).
p7(1.050, 0.780) reached. Going to p8(1.050, 0.490).
p8(1.050, 0.490) reached. Going to goal(0.950, 0.370).
goal(0.950, 0.370) reached. Final goal reached!!
```

- **Test 4:**

```
start(0.440, 1.010) reached. Going to p2(0.370, 1.050).
p2(0.370, 1.050) reached. Going to p1(0.170, 0.970).
p1(0.170, 0.970) reached. Going to p3(0.170, 0.780).
p3(0.170, 0.780) reached. Going to p4(0.510, 0.780).
p4(0.510, 0.780) reached. Going to p17(0.510, 0.600).
p17(0.510, 0.600) reached. Going to p12(0.710, 0.410).
p12(0.710, 0.410) reached. Going to p10(0.680, 0.150).
p10(0.680, 0.150) reached. Going to goal(0.780, 0.110).
goal(0.780, 0.110) reached. Final goal reached!!
```

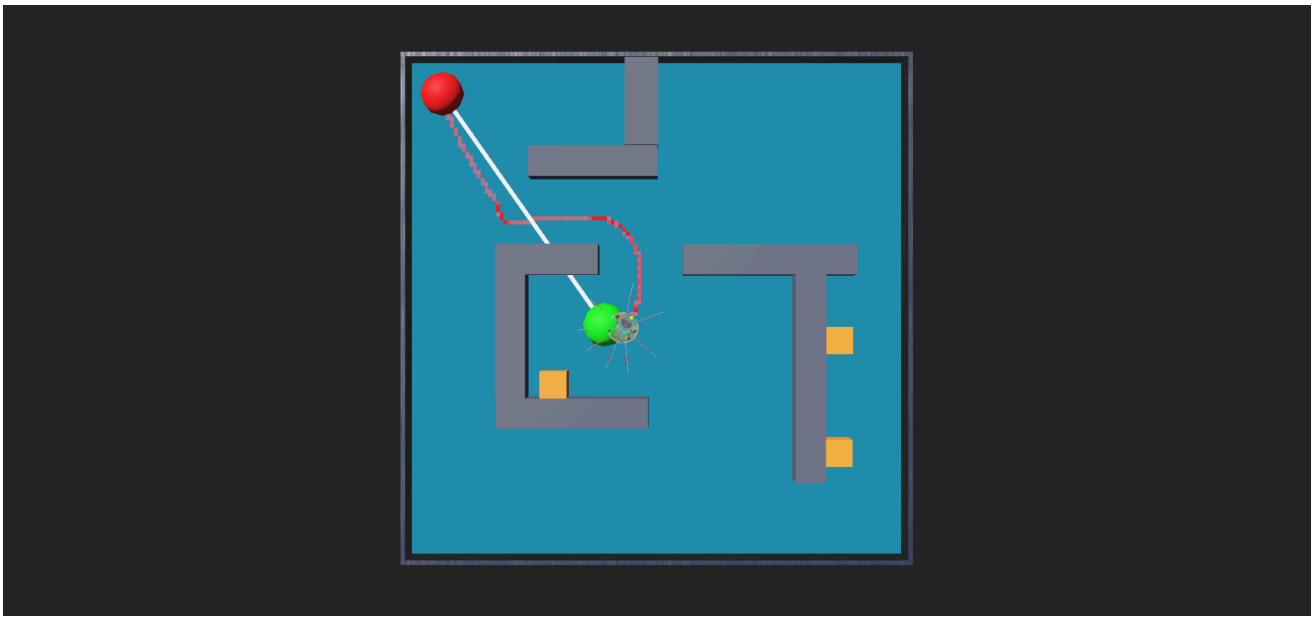
Question 2

a)

We will be using the same APF controller from the previous project. Below, we provide a simple description of our APF controller, please refer to the report of project 1 for more details.

The APF consists of two behaviors: attraction which will drive the robot into the next waypoint, and repulsion which will provide low-level collision avoidance from reactive layer level. The attraction vector is simply found by subtracting the goal position vector from the current position vector. The repulsive vectors are calculated by converting the distance sensor readings into vectors using the angles at which they are situated. Moreover, the speed of the robot is reversely proportional to the turning angle. If the turning angle is zero, meaning the robot is moving in a straight line, the speed is step to max. On the other hand, if the turning angle is near 90 degrees, meaning that the robot is making a very sharp turn, the speed is almost zero. This makes the robot move as fast as possible in straight lines, and slow down on turns.

b)



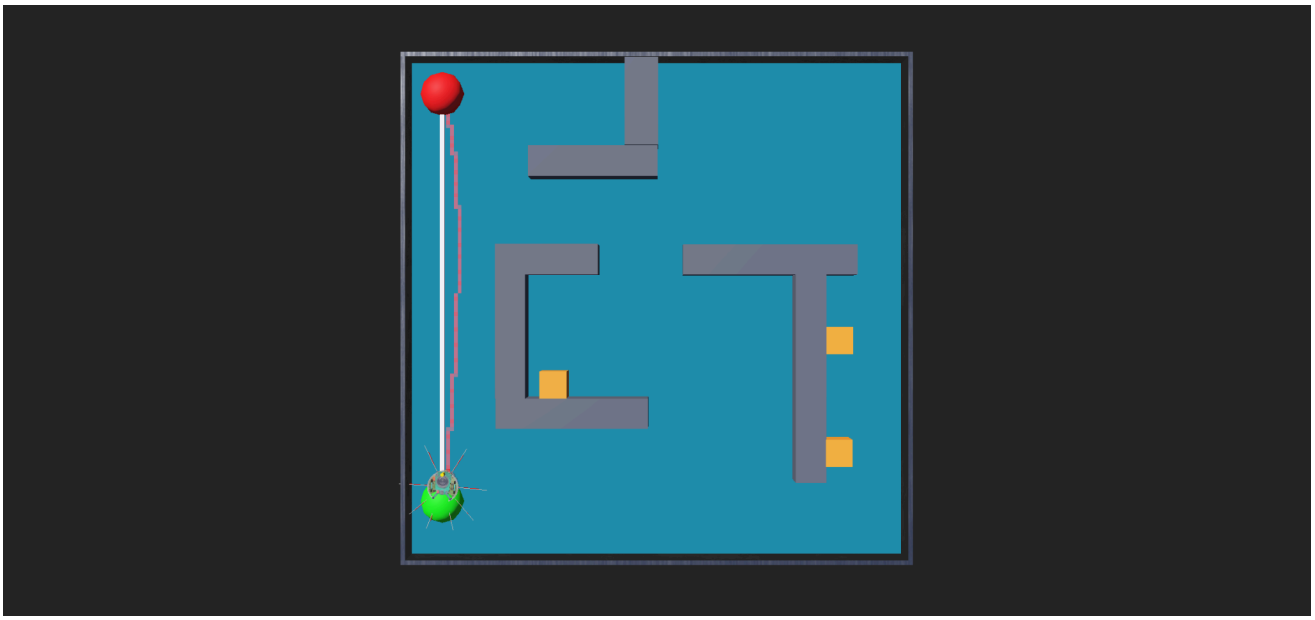
The robot successfully reaches the goal destination with a reactive layer only in this simple case. The robot starts by driving itself directly into the goal, but it faces an obstacle and avoids it correctly until it reaches the opening and then it can be attracted to the goal.

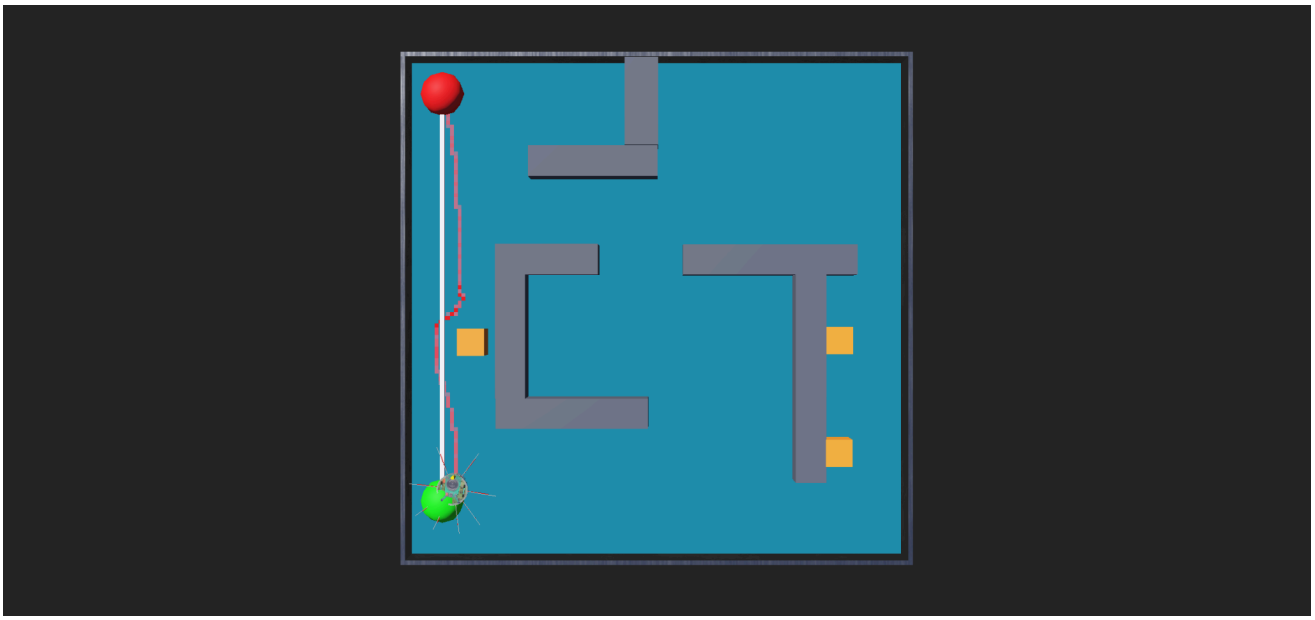
This test was done by replacing the waypoints dictionary with an empty one:

```
graph = Graph(
    adjacency_graph={}, # Empty map. start and goal passed in the next arguments
    start=start,
    goal=goal,
    cost_function=cost_func,
    heuristic_function=heuristic_func,
)
```

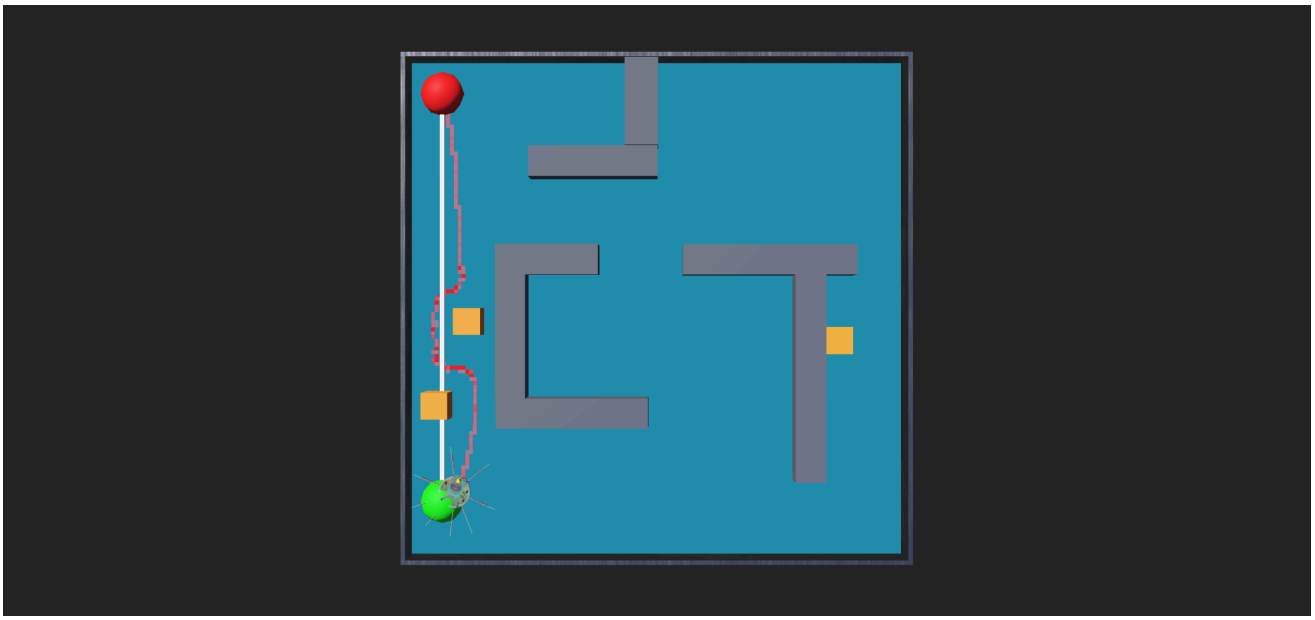
c)

Below is the robot's trajectory with and without obstacles:





As expected the robot reacts correctly to the obstacles. When the sensors feel an obstacle on the left, the repulsive force pushes the robot to the right and vice versa. We even made it more challenging for the robot by placing two close obstacles, one of them placed on the (white) path itself.



Note how the robot managed to make a very sharp turn (almost 90 degrees) to avoid the second obstacle, while still re-correcting its path to reach the goal successfully.

Question 3

a)

```
class APFController:
    def __init__(
        self,
        robot: Robot,
        deliberative_layer: DeliberativeLayer,
        *,
        distance_to_goal_threshold: float = 0.05,
    ) -> None:
        self.__robot = robot
        self.__distance_threshold = distance_to_goal_threshold
        self.__final_goal_reached: bool = False
        self.__deliberative_layer = deliberative_layer
        self.__destination =
self.__deliberative_layer.get_next_waypoint()
```

To combine the reactive layer (we are calling it `APFController`) with the deliberative layer we are using dependency injection. We are injecting an instance of the `DeliberativeLayer` to the constructor of the `APFController` class. This way, the `APFController` can use the deliberative layer to get the next waypoint by simply calling `deliberative_layer.get_next_waypoint()`.

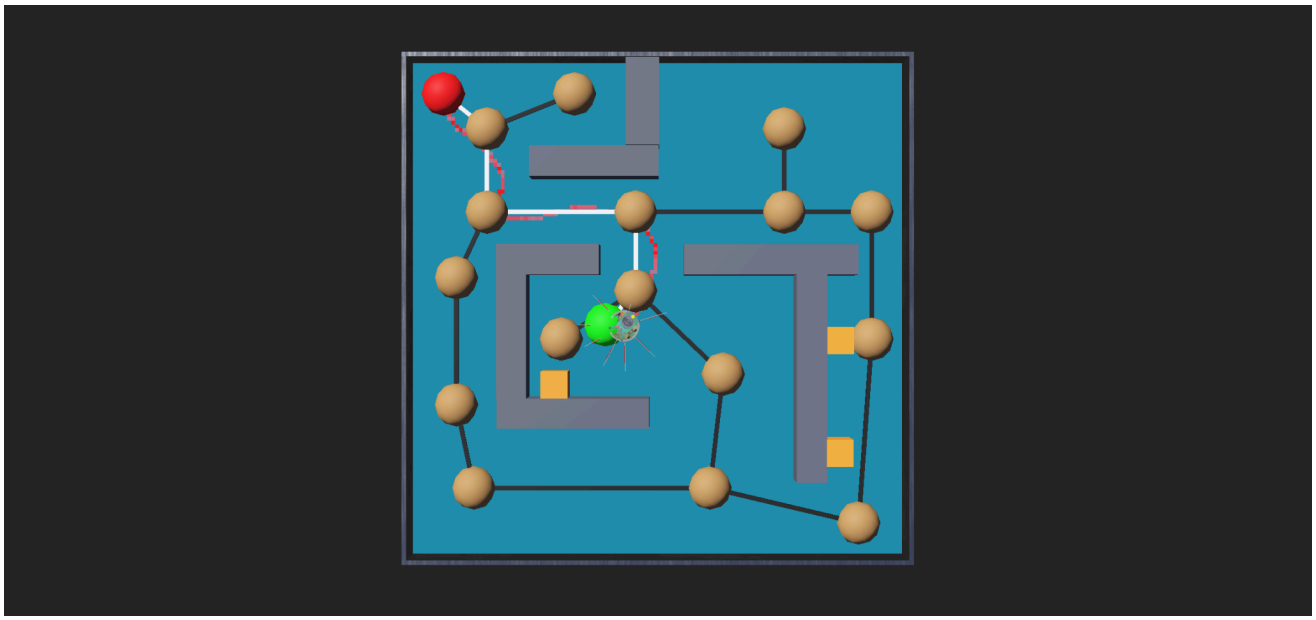
Extra, a software engineering note: one might think injecting the whole `DelivarativeLayer` object is too much and might lead to misusing the internal attributes of the `DelivarativeLayer` by the other class. However, the internal attributes of the `DelivarativeLayer` class are made private. So in the normal use case[6], these internal attributes will not be misused by the other class.

[6] Python does not support real private and public attributes, so the user can still access private data if they really want to.

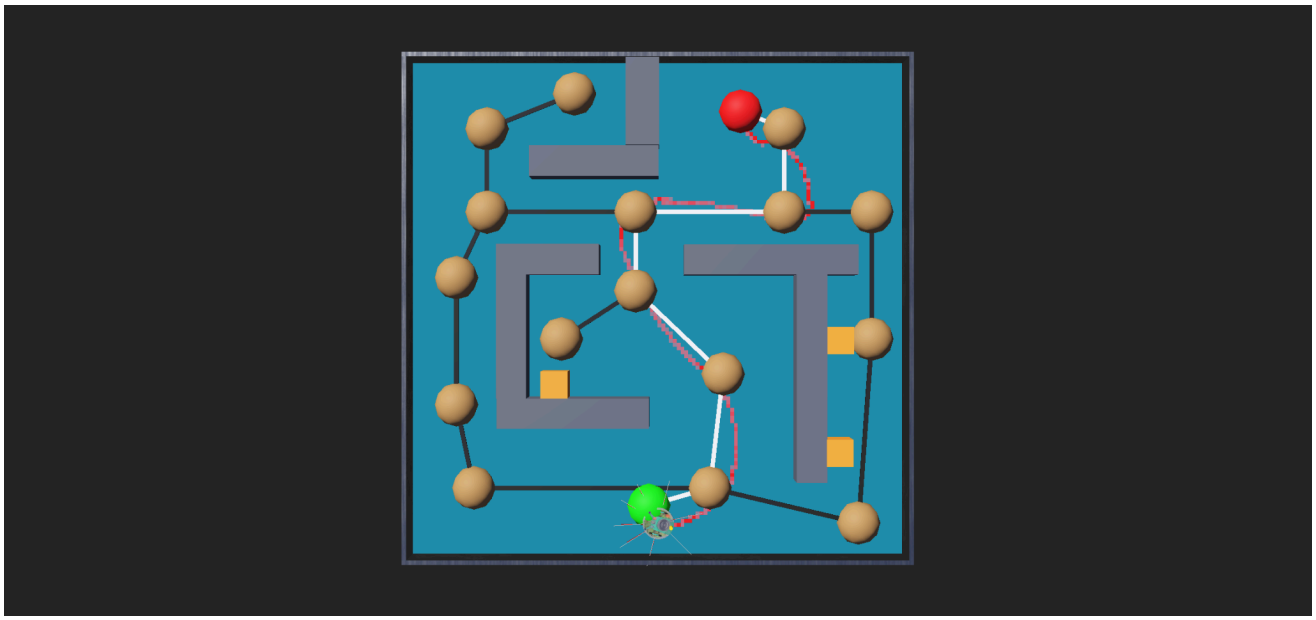
b)

Below, we show the trajectories of the robot when the reactive and deliberative layers are combined. Although the environment is static, the reactive deliberative layer alone is not enough due to control errors. Hence, the reactive layer helped reach the goal without collisions. We can see the reactive controller didn't follow the path exactly, which is the expected behavior for an APF controller.

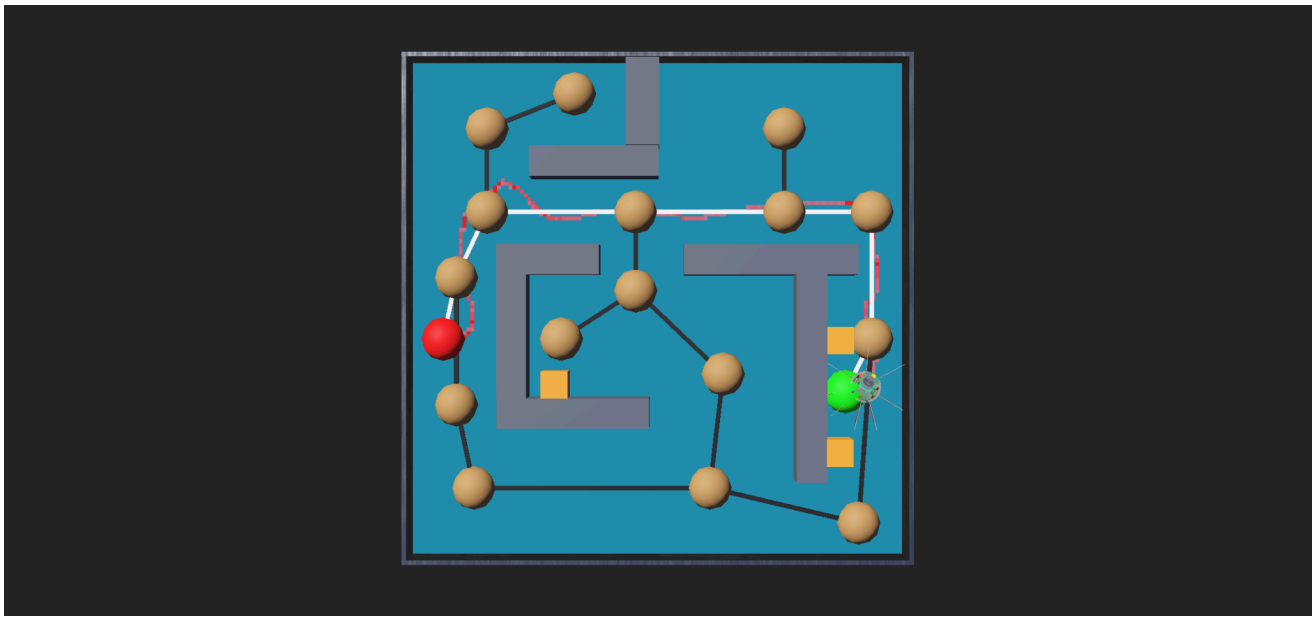
Test 1



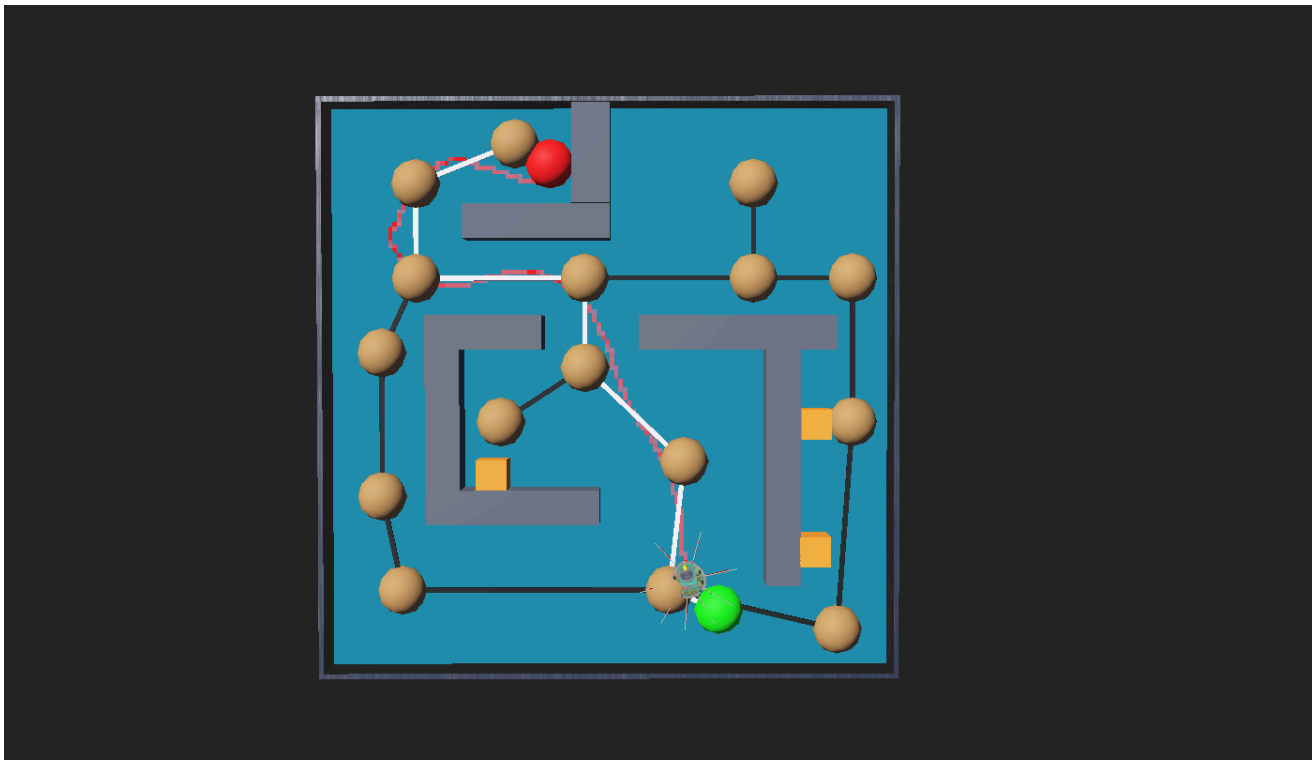
Test 2



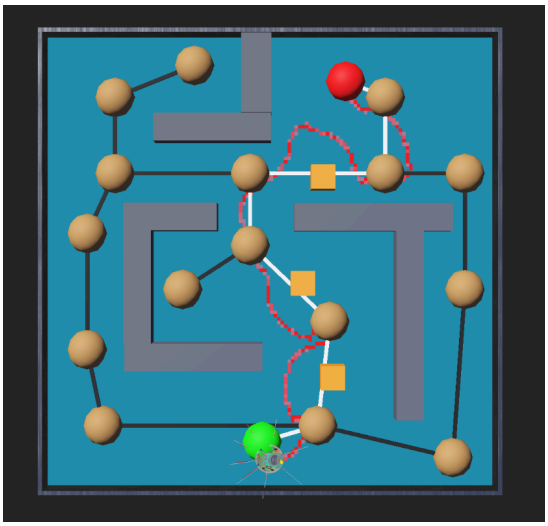
Test 3

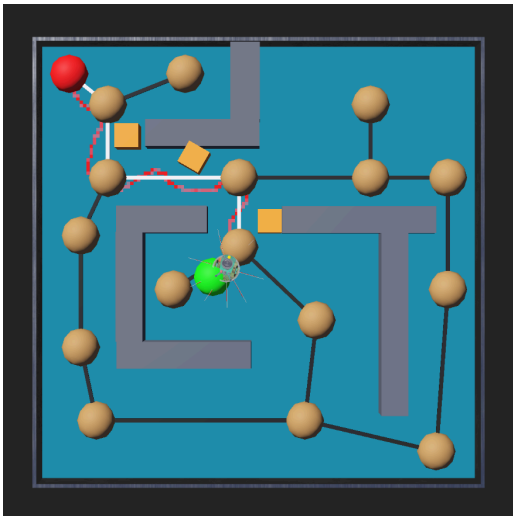


Test 4



c)





Like previous results, the robot, using the reactive layer, manages to avoid obstacles while still correcting the path to reach the next waypoint.

Below is the output of the simulation shown in the left picture above. Note that the controller prints the path before moving the robot, and then prints the next waypoint in the planned path as the robot moves.

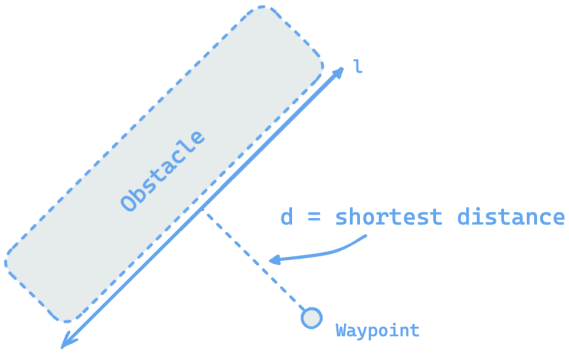
```
Console - All
Starting simulation...
Test ID: test2
Path type: shortest
Path:
 start(0.750, 1.010) --> p5(0.850, 0.970) --> p6(0.850, 0.780) --> p4(0.510, 0.780) -->
p17(0.510, 0.600) --> p12(0.710, 0.410) --> p10(0.680, 0.150) --> goal(0.540, 0.110)
INFO: 'supervisor' controller exited successfully.
start(0.750, 1.010) reached. Going to p5(0.850, 0.970).
p5(0.850, 0.970) reached. Going to p6(0.850, 0.780).
p6(0.850, 0.780) reached. Going to p4(0.510, 0.780).
p4(0.510, 0.780) reached. Going to p17(0.510, 0.600).
p17(0.510, 0.600) reached. Going to p12(0.710, 0.410).
p12(0.710, 0.410) reached. Going to p10(0.680, 0.150).
p10(0.680, 0.150) reached. Going to goal(0.540, 0.110).
goal(0.540, 0.110) reached. Final goal reached!!
INFO: 'main' controller exited successfully.
```

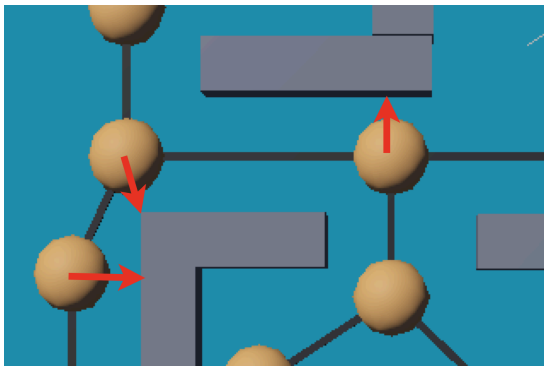
d)

Path cost

In order to produce the safest path, the path cost (g-cost) was changed. The g-cost at any node is now the inverse of the minimum distance to the nearest obstacle. So when the node is close to an obstacle the cost is high (avoid) and when it is far from any obstacle the cost is low (preferred). To find the closest distance from a given node to any obstacle we first represented the obstacles in the code as rectangles by saving the top left and bottom right coordinates in a `Rectangle` object. The reason behind this is that all obstacles on the map are either horizontal (0°) or vertical (90°) rectangles. Additionally, all `Rectangle` objects are stored in an `ObstaclesMap` object which offers an interface to compute the closest distance.

To get the closest distance we call `get_closest_obstacle_distance()` which in turn calls the `__get_perpendicular_distance()` on all 4 sides of all obstacles which calculates the point-to-line segment distance, as shown in the pictures below. The minimum is taken and the inverse is used as the g-cost.





```
class ObstaclesMap:

    def __init__(self, rectangular_obstacles: List[Rectangle]) -> None:

        """Representation of the rectangular obstacles on the map"""

        self.__obstacles: List[Tuple[Point, Point, Point, Point]] = []

        # storing the four points of the rectangle

        for rectangle in rectangular_obstacles:

            top_left = rectangle.top_left

            bottom_right = rectangle.bottom_right

            top_right = Point(bottom_right.x, top_left.y)

            bottom_left = Point(top_left.x, bottom_right.y)

            self.__obstacles.append((top_left, top_right,

                                     bottom_right, bottom_left))


    def __get_perpendicular_distance(self, point: Point, line: Line) -> float:

        A = point.x - line.start.x

        B = point.y - line.start.y

        C = line.end.x - line.start.x

        D = line.end.y - line.start.y

        dot = A * C + B * D

        len_sq = C * C + D * D

        param = -1

        if len_sq != 0:  # in case of 0 length line

            param = dot / len_sq

        if param < 0:

            xx, yy = line.start.x, line.start.y

        elif param > 1:

            xx, yy = line.end.x, line.end.y

        else:
```

```
            xx = line.start.x + param * C
            yy = line.start.y + param * D
        dx = point.x - xx
        dy = point.y - yy
        return np.sqrt(dx**2 + dy**2)


    def get_closest_obstacle_distance(self, point: Waypoint, _) -> float:
        """Calculates the shortest distance from the point to
           any edge of the obstacle rectangles."""
        min_distance = float("inf")
        for obstacle in self.__obstacles:
            # Calculate distance to each edge of the rectangle
            for i in range(4):
                start_vertex = obstacle[i]
                end_vertex = obstacle[(i + 1) % 4]
                line = Line(start_vertex, end_vertex)
                dist = self.__get_perpendicular_distance(point, line)
                if dist < min_distance:
                    min_distance = dist
        return 1.0 / min_distance if min_distance != float("inf") else 1e-6
```

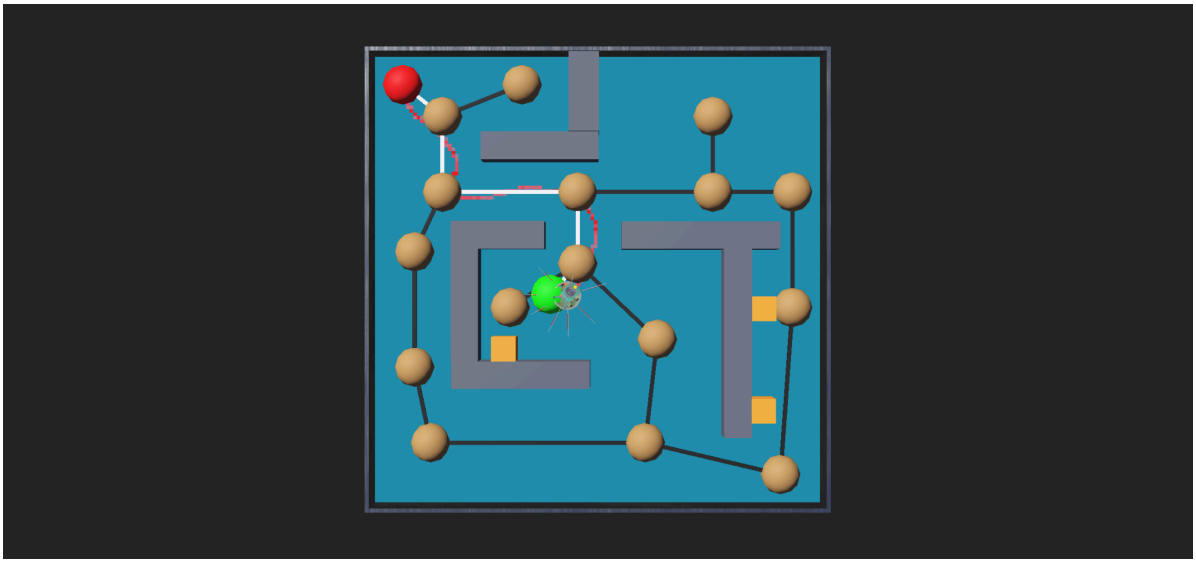
```
obstacle_map = ObstaclesMap(
            [
                # the 7 rectangular obstacles
                Rectangle(Waypoint(0.49, 1.12), Waypoint(0.56, 0.92)),
                Rectangle(Waypoint(0.27, 0.92), Waypoint(0.56, 0.85)),
                Rectangle(Waypoint(0.208, 0.706), Waypoint(0.438, 0.636)),
                Rectangle(Waypoint(0.208, 0.636), Waypoint(0.268, 0.356)),
                Rectangle(Waypoint(0.198, 0.356), Waypoint(0.538, 0.296)),
                Rectangle(Waypoint(0.619, 0.698), Waypoint(1.01, 0.636)),
                Rectangle(Waypoint(0.87, 0.636), Waypoint(0.936, 0.172)),
                # the 4 map walls
                Rectangle(Waypoint(-0.005, 1.12), Waypoint(0, 0)),
                Rectangle(Waypoint(-0.005, 1.13), Waypoint(1.12, 1.12)),
                Rectangle(Waypoint(1.12, 1.12), Waypoint(1.13, -0.0142)),
                Rectangle(Waypoint(-0.0149, -0.005), Waypoint(1.13, -0.014)),
            ]
        )
```

Heuristics

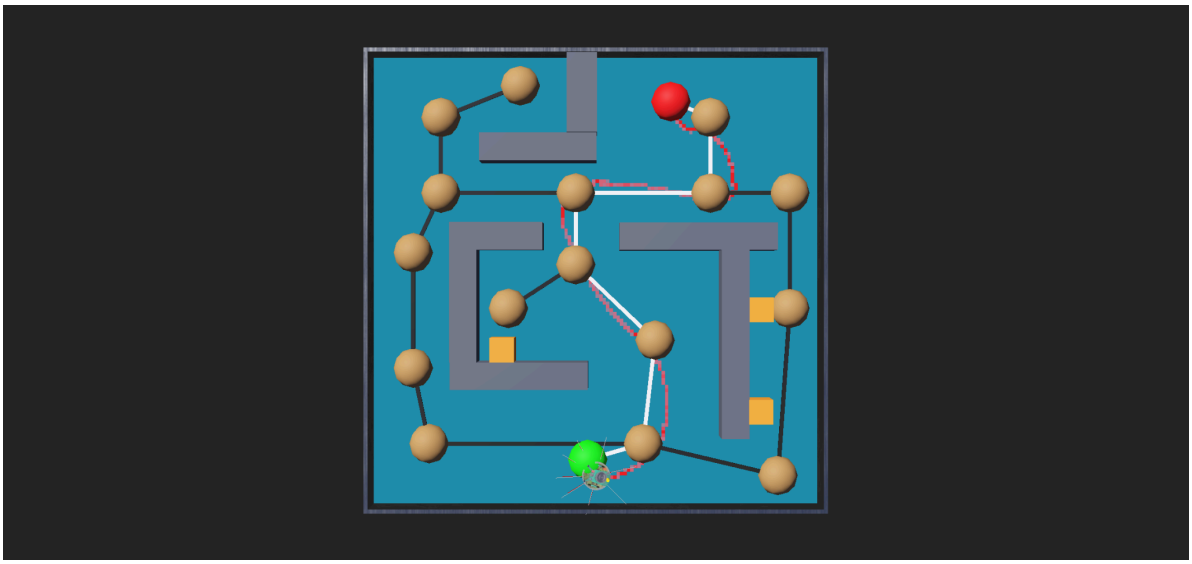
Heuristic function: we implemented a simple heuristic function that just returns the same cost used in the cost function (the inverse of the distance to the closest obstacle). The logic behind this is that in our map the same corridor has two to three nodes. These nodes approximately have the same cost metric, so if we use the same cost metric (non-cumulative) for the heuristic which represents the predicted *remaining* cost, we are predicting what the future cost will look like and hence guiding the search and since we assume the robot is at the beginning of a corridor, thus there is only one more node remaining in the corridor so we set the h-cost equal to the g-cost (non-cumulative). This is not the best heuristic for sure since it is map dependent, but a search with heuristic is always better than without as long as it does not overestimate. We are assuming this will never overestimate if the path is long enough. The results from using this heuristic were compared to the results without using the heuristic, and we got the same results in both. Even if the next node does not have the same cost as the current node, the total path cost will still be higher than the heuristic if the path is long and thus not overestimating.

Below, we show the generated paths of the four test cases using the safest path.

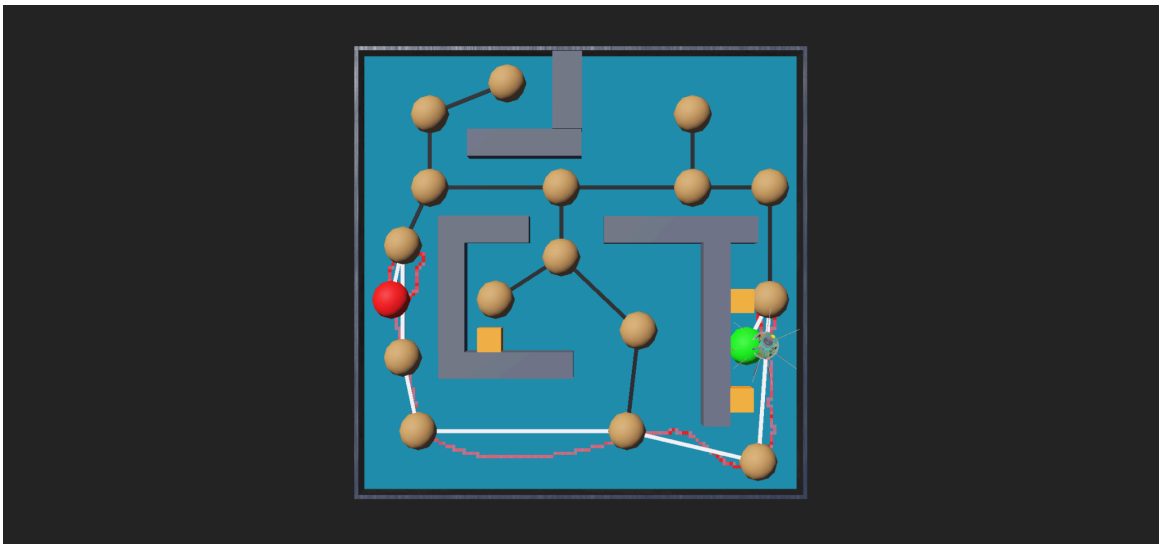
Test 1



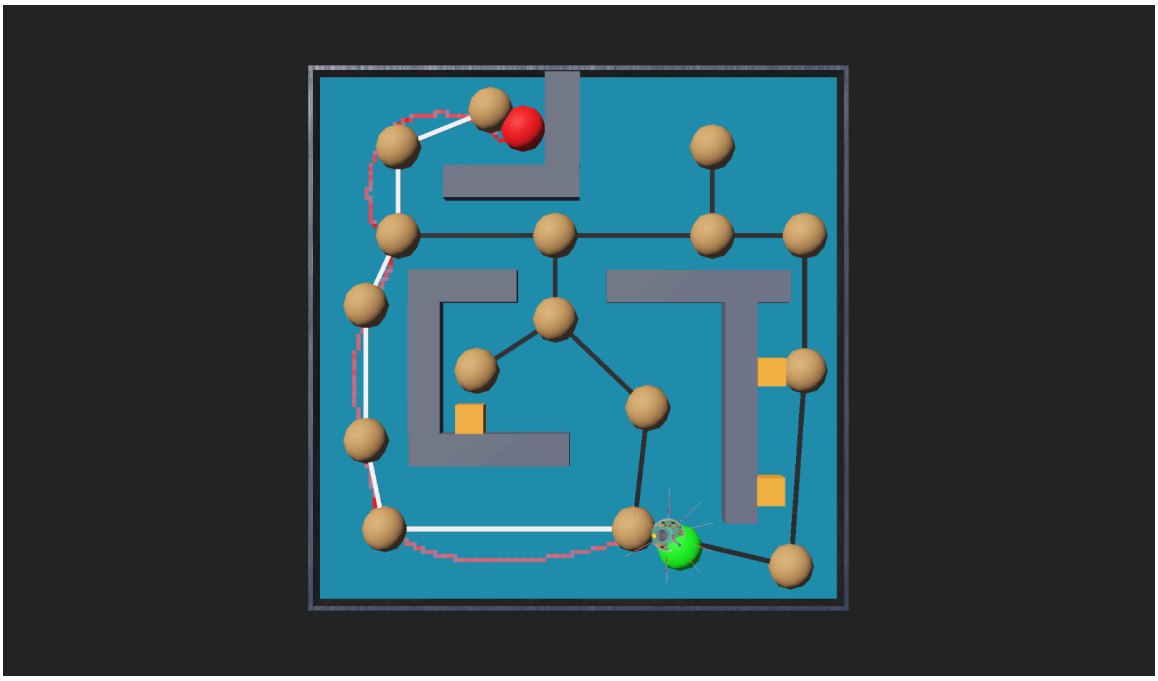
Test 2



Test 3

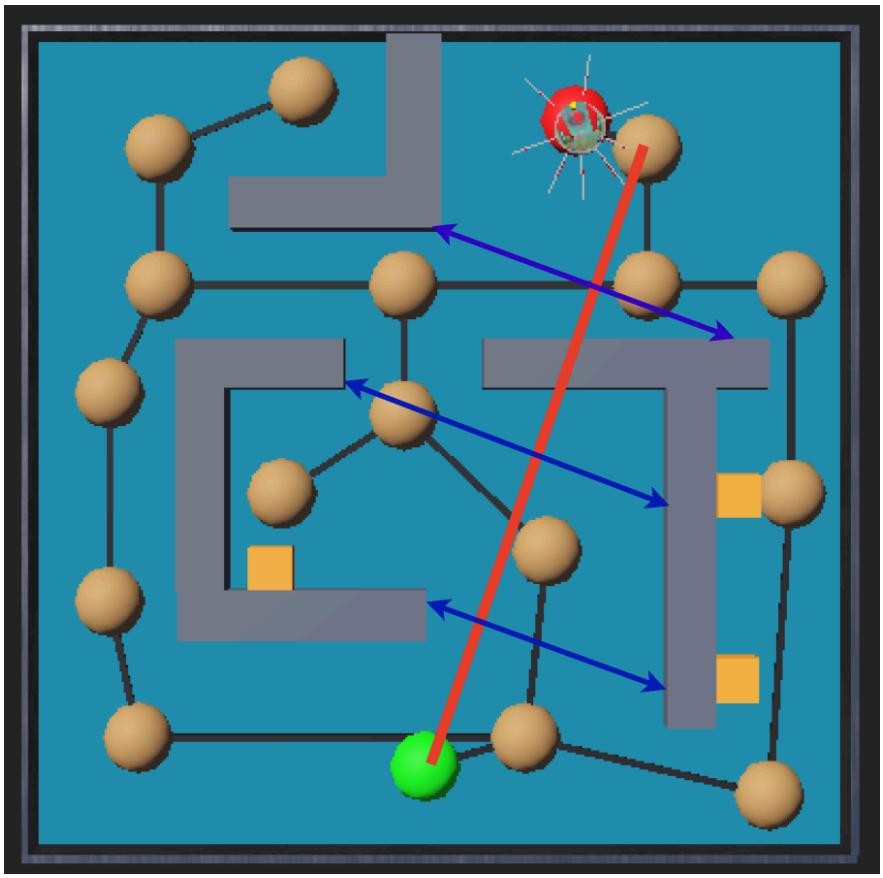


Test 4



For test 1 and 2, the safest path happens to be the shortest path too (check Q3b). However, the safest path in test 3 and 4 are not the same as the shortest path. For test 3 and 4, the robot preferred to go to the bottom corridor, which makes sense since this corridor is the widest area in the whole map, meaning that the robot will be very far from the walls.

Another heuristic that was designed *but not implemented* is described hereafter. The idea is that at any given node we draw a straight line to the goal node and on that line we choose x points that we expand perpendicular to the straight line (in both directions) until we hit an obstacle[7] and we sum up the minimum distances to the obstacles across the x points. However, there are two main issues with this approach. First, x is a hyperparameter that should be experimentally determined. Second, admissibility (not overestimating) depends on the hyperparameter x, if x is too large then it may overestimate the actual sum of minimum distances on the path (g-cost) and lead to suboptimal paths. An example of this heuristic with x = 3 is shown below



[7] The points are expanded in both directions perpendicular to the line and the minimum of them is chosen