CDE2310 Navigation Algorithm

Group 2

1 Introduction

The navigation system of our robot is built upon three foundational pillars: locating the target, planning an optimal path, and executing movement to reach the target. Each of these pillars is supported by dedicated algorithmic strategies, which are described in the following sections.

2 Localization

Before diving into the three foundational pillars, there is a fundamental problem that must be addressed: *How can we determine the robot's location on the map?* To answer this, the team decided to use the `tfBuffer` provided by ROS2. The implementation of this can be found in the **Appendix A**.

3 Target Finding

3.1 Initial Approach: Greedy Based on Total Distance

The team's first approach to target identification applies a greedy heuristic. The heuristic aims to search a point that maximizes the sum of Euclidean distances to all previously visited points.

Formally, let $\mathbf{v}$ be the array of all visited points with size k, and let $\mathbf{v}_i$ denote the i-th visited point where $1 \leq i \leq k$. The goal is to find a point $\mathbf{b}$ that maximizes the expression:

$$\sum_{i=1}^{k} \text{distance}(\mathbf{b}, \mathbf{v}_i)$$

The implementation of this approach is provided in **Appendix B**.

However, after several testing, the team realizes edge cases where the greedy heuristic is not efficient. Consider the edge case below!

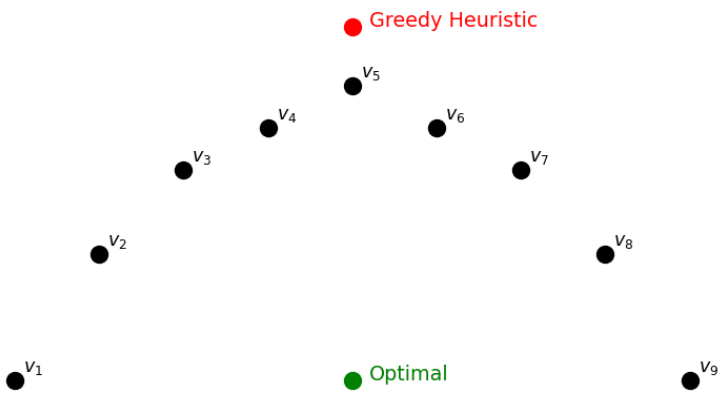

Figure 1: Edge Case for Greedy Based on Total Distance

In this edge case, the greedy heuristic detects the red point as the optimal target even though the optimal point to explore is the green point.

3.2 Revised Approach: Greedy Based on Minimum Distance

Upon examining the edge cases of the first heuristic, the team refined our approach. The revised strategy focuses on finding a point that maximizes the minimum Euclidean distance to any of the previously visited points.

Formally, with the same notation as above, the objective becomes finding a point $\mathbf{b}$ that maximizes:

$$\min_{i=1}^{k} \text{distance}(\mathbf{b}, \mathbf{v}_i)$$

The implementation of this approach is provided in **Appendix C**.

However, after many trials, this greedy approach also fails in several edge cases. Consider the scenario below.

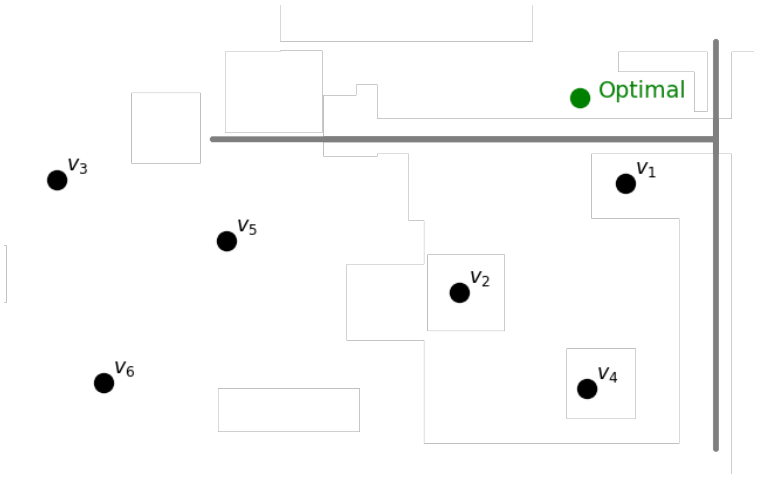

Figure 2: Edge Case for Greedy Based on Minimum Distance

In the Euclidean distance perspective, the green point is located near $\mathbf{v}_1$. Therefore, the greedy heuristic will treat these two points as close to each other, even though in reality it does not. This suggest that this greedy heuristic need to be improved, especially the limitation of using Euclidean distance.

3.3 Final Approach: Multi-source Dijkstra

To address the limitations of using Euclidean distance in complex terrains, the team's final approach utilizes a cost-based method. Specifically, the team applies the multi-source Dijkstra algorithm, treating all visited points as sources. This results in a cost map that reflects the minimum traversal cost to a point from the closest visited point. In order to make the robot explores as far as possible, the point which has highest cost will then be chosen.

The implementation of this approach is provided in **Appendix D**.

3.4 Minor Improvement

Due to the map update delay discussed in Chapter 7, the robot may occasionally collide with obstacles that have not yet been reflected in the updated map, especially when exploring distant areas. To mitigate this risk, the team introduce a new threshold called DISTANCE_THRESHOLD. This ensures that the robot avoids exploring points with a cost higher than DISTANCE_THRESHOLD, thereby reducing the likelihood of encountering unupdated or unknown obstacles.

The implementation of the algorithm after this improvement is provided in **Appendix E**.

4 Path Finding

4.1 Initial Approach: Plain A* Search Algorithm

The team's first approach on finding the shortest path to the target is to use the plain A* Search Algorithm. The implementation of this approach can be found in **Appendix F** (for helper functions) and **Appendix G** (for A* Search implementation).

However, after analyzing the generated path, the team realized that the selected points were too close to the wall, making it likely that the robot would crash if it followed the path.

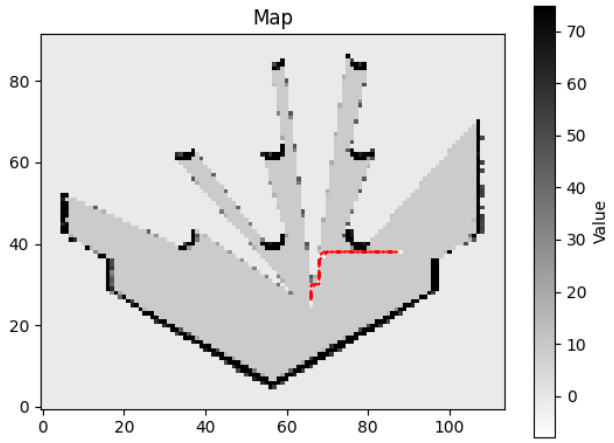

Figure 3: Path produced (in Gazebo Simulation)

4.2 Final Approach: A* Search Algorithm with Wall Penalty

To avoid potential crashes, the team decided to add a wall penalty mechanism that will add up to the A* heuristic when it is too close to the wall. By this, the algorithm will penalize locations which are too close to the wall, avoiding them to be used. The implementation of this approach can be found in **Appendix F** (for helper functions) and **Appendix H** (for A* Search implementation).

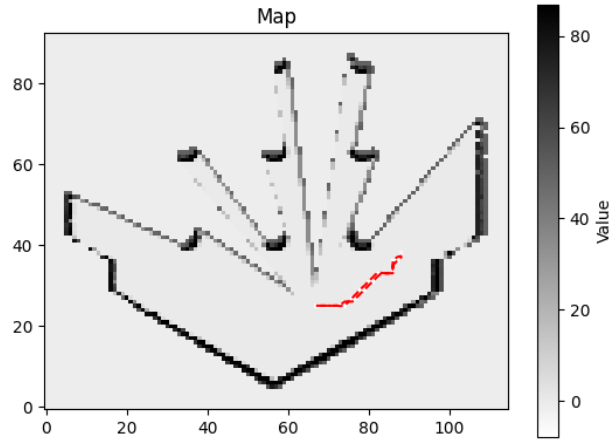

Figure 4: Path produced when wall penalty is applied (in Gazebo Simulation)

5 Movement Execution

5.1 Initial Approach: Point-to-Point Movement Strategy

The team's initial approach on movement execution was to make the robot follow the path generated by the A* Search Algorithm, navigating from point to point. The implementation of this method is provided in **Appendix I**. However, in practice, the robot's movement is inconsistent, almost as if it were "jumping" between locations, due to localization inaccuracies.

5.2 Final Approach: Cluster-to-cluster Movement Strategy

To address the localization issue, the team proposed an alternative movement strategy: navigating the robots from cluster to cluster. These cluster points are selected from the path generated by the A* Search Algorithm, as discussed in Chapter 4.2. The implementation details of the clustering approach can be found in **Appendix J**.

Then, to account for localization inaccuracy, the team introduced a tolerance margin, meaning the robot does not need to reach an exact coordinate. As long as it remains within a certain distance of the target point, it is considered a successful arrival. The implementation of this movement strategy can be found in **Appendix K** (for helper functions) and **Appendix L** (for movement implementation).

6 Heat Pursuit Movement

When heat is detected and the distance between the current point and the closest shooting area is higher than SHOOTING_AREA_THRESHOLD, an interrupt routine is triggered to process the event. In a separate program called `elec_nodes.py` (code can be found by clicking this link), nearby heat locations are published regularly via a ROS2 topic named `heat_location`. The heat information can be one of five types: `left`, `right`, `forward`, `ok`, and `null`. Based on the received signal, the robot will respond accordingly:

- `null` : No interrupt is triggered
- `right` : Stop the robot. Rotate the robot to the right by HEAT_ROTATE_ANGLE
- `left` : Stop the robot. Rotate the robot to the left by HEAT_ROTATE_ANGLE
- `forward` : Stop the robot. Move the robot forward
- `ok` : Stop the robot. Activate the shooting mechanism

The implementation of this interrupt routine can be found in **Appendix O**. To see in detail where this routine is ran, see the full implementation of move_through_path function here.

7 Map Update Delay

Initially, the team's strategy for updating the map within the callback function was to update it every time the callback was triggered (implementation can be found in **Appendix M**). However, this approach led to a problem. During movement execution, the robot heavily relies on row and column indices. As the map expands, these indices also change, causing inconsistencies that affect the robot's movement.

To address this issue, the team introduced a control flag variable called `can_update`. This flag determines whether the map should be updated within the callback function. With this modification, the map is only updated when `can_update` is explicitly enabled, ensuring consistent behavior during movement execution. The value of `can_update` is managed within the move_through_path function (can be found in **Appendix L**). The implementation of this can be found in **Appendix N**).

8 Points Storing Mechanism

Recall the earlier observation: once the map is updated, the previously calculated row and column indices may no longer be valid. This is because the grid layout shifts with changes in the map origin. To address

this, we made a slight modification in how points are stored in the visited points set and the shooting area set.

Consider the code snippet below, which computes the row and column indices:

```
grid_x = round((self.cur_pos.x - self.map_origin.x) / self.map_res)  # column in
    numpy
grid_y = round((self.cur_pos.y - self.map_origin.y) / self.map_res)  # row in
    numpy

self.currow = self.map_width - 1 - grid_x
self.curcol = self.map_height - 1 - grid_y
```

Notice that while `cur_pos` remains unchanged, the resulting indices (`currow` and `curcol`) vary depending on `map_origin`. Therefore, instead of storing `currow` and `curcol` directly in the visited points and shooting area sets, we store the actual `cur_pos`. This change is also reflected when constrasting the initial (**Appendix M**) and final (Appendix **Appendix N**) implementation of the callback function.

Later, whenever we need to perform operations involving these sets, we convert the stored positions into row and column indices as needed. This approach has already been applied in the `find_target` function for the visited point set (see **Appendix E**) and the heat interrupt routine for the shooting area set (see **Appendix O**).

9 Parameter Descriptions

The navigation code relies heavily on these parameters. Tune these parameters to achieve the best performance. Understand what these values are and tune the parameters by making observation during testing.

1. SHOOTING_AREA_THRESHOLD:

   Let x represent the distance from the current location to the nearest shooting area. This value is compared against a SHOOTING_AREA_THRESHOLD threshold to determine whether the heat interrupt should be processed:

   $$\begin{cases} x < \text{SHOOTING\_AREA\_THRESHOLD} & : \text{Ignore the heat interrupt} \\ x \geq \text{SHOOTING\_AREA\_THRESHOLD} & : \text{Proceed to check the heat interrupt} \end{cases}$$

2. RESET_VISITED_POINTS_THRESHOLD:

   This threshold controls the maximum number of points that can be stored in the visited points set. Once the size of the set exceeds this threshold, the oldest point will be removed.

   The main purpose of this variable is to reduce computational complexity when the visited points set becomes too large. However, if the threshold is set too low, the robot may repeatedly explore the same areas, as it will forget previously visited points too quickly.

   Suggestion: Observe the chosen path from the map and increase the threshold if the robot keeps exploring the same area too fast.

3. HEAT_ROTATE_ANGLE:

   This threshold controls how many degrees to turn left or right when heat interrupt is triggered and heat is detected on left or right.

   Suggestion: If the robot turns to the left or right too much when heat interrupt is triggered, lower down the value of this threshold.

4. WALL_THRESHOLD

Let x represent a value from the LiDAR-generated map. This value is compared against a threshold to determine how each point is interpreted:

$$\begin{cases} x < \text{WALL\_THRESHOLD} : \text{Empty space} \\ x \geq \text{WALL\_THRESHOLD} : \text{Obstacle} \end{cases}$$

The threshold WALL_THRESHOLD distinguishes between empty space and obstacles based on LiDAR readings. This threshold is only used in the isValid function that can be found in **Appendix F**.

5. DISTANCE_THRESHOLD

   This threshold controls the highest cost that can be chosen to be explored. This threshold is only used in the find_target function that can be found in **Appendix E**. Refer to Chapter 3.4 for more detailed explanation.

6. wall_penalty

   This threshold defines the maximum allowed row and/or column distance for a point to be considered penalized. This threshold is used in the astar that can be found in **Appendix H** and find_target function that can be found in **Appendix E**.

   Alert: This threshold is highly sensitive. If set too high, certain areas of the map may remain unexplored because A* Search is unable to find a path to it. If set too low, the robot may frequently crash into obstacles. Therefore, it is important to tune this value carefully.

7. cluster_distance

   This threshold defines the minimum distance that separate any two adjacent cluster points. It is used only in the cluster_path function, which can be found in **Appendix J**.

8. localization_tolerance: controls the tolerance margin that is described in Chapter 5.2.
9. rotatechange: controls the robot's angular speed.
10. speedchange: controls the robot's linear speed.

A Localization

```
# Declared in the init function of the class
self.tfBuffer = tf2_ros.Buffer(cache_time=rclpy.duration.Duration(seconds=5.0))
self.tfListener = tf2_ros.TransformListener(self.tfBuffer, self)

# Implemented in the map callback funtion
try:
    trans = self.tfBuffer.lookup_transform(
        'map', 'base_link',
        rclpy.time.Time(),
        timeout=rclpy.duration.Duration(seconds=1)
    )
except (LookupException, ConnectivityException, ExtrapolationException) as e:
    print(e)
    return

self.cur_pos = trans.transform.translation
```

B Greedy Based on Total Distance Implementation

```
def find_target(self):
    dis = -math.inf
    targetrow, targetcol = -1, -1
    num_rows, num_cols = self.cur_map.shape
    for idxrow in range(num_rows):
        for idxcol in range(num_cols):
            too_close_to_wall = False
            for penaltyx in range(-wall_penalty, wall_penalty + 1):
                if too_close_to_wall:
                    break
                for penaltyy in range(-wall_penalty, wall_penalty + 1):
                    next_row = idxrow + penaltyx
                    next_col = idxcol + penaltyy
                    if not self.isValid(next_row, next_col):
                        too_close_to_wall = True
                        break
            if too_close_to_wall:
                continue
            if self.cur_map[idxrow][idxcol] == 0:
                cur_dis = 0
                for x, y in self.visited_points:
                    cur_dis += (idxrow - x) ** 2 + (idxcol - y) ** 2
                if dis < cur_dis:
                    dis = cur_dis
                    targetrow = idxrow
                    targetcol = idxcol
    return (targetrow, targetcol)
```

C Greedy Based on Minimum Distance Implementation

```
def find_target(self):
    dis = -1
    targetrow, targetcol = -1, -1
    num_rows, num_cols = self.cur_map.shape
    self.get_logger().info('Finding Target')
    for idxrow in range(num_rows):
        for idxcol in range(num_cols):
            too_close_to_wall = False
            for penaltyx in range(-wall_penalty, wall_penalty + 1):
                if too_close_to_wall:
                    break
                for penaltyy in range(-wall_penalty, wall_penalty + 1):
                    next_row = idxrow + penaltyx
                    next_col = idxcol + penaltyy
                    if not self.isValid(next_row, next_col):
                        too_close_to_wall = True
                        break
            if too_close_to_wall:
                continue
            if self.cur_map[idxrow][idxcol] == 0:
                min_dis = math.inf
                for x, y in self.visited_points:
                    grid_x = round((x - self.map_origin.x) / self.map_res)
                    grid_y = round(((y - self.map_origin.y) / self.map_res))
                    convertx = self.map_width - 1 - grid_x
                    converty = self.map_height - 1 - grid_y
                    min_dis = min(min_dis, (idxrow - convertx) ** 2 + (idxcol -
                        converty) ** 2)
                if dis < min_dis:
                    dis = min_dis
                    targetrow = idxrow
                    targetcol = idxcol
    return (targetrow, targetcol)
```

D Multi-source Dijkstra Implementation

```
def find_target(self):
    num_rows, num_cols = self.cur_map.shape
    cost_map = [[-1 for _ in range(num_cols)] for _ in range(num_rows)]
    for x, y in self.visited_points:
        grid_x = round((x - self.map_origin.x) / self.map_res)
        grid_y = round(((y - self.map_origin.y) / self.map_res))
        convertx = self.map_width - 1 - grid_x
        converty = self.map_height - 1 - grid_y
        cost_map[convertx][converty] = 0
    points = []
    for x, y in self.visited_points:
        grid_x = round((x - self.map_origin.x) / self.map_res)
        grid_y = round(((y - self.map_origin.y) / self.map_res))
        convertx = self.map_width - 1 - grid_x
        converty = self.map_height - 1 - grid_y
        heapq.heappush(points, (0, convertx, converty))
    while points:
        curcost, currow, curcol = heapq.heappop(points)
        for idx in range(4):
            nextrow = currow + dr[idx]
            nextcol = curcol + dc[idx]

            if not self.isValid(nextrow, nextcol):
                continue

            too_close_to_wall = False
            for penaltyx in range(-wall_penalty, wall_penalty + 1):
                if too_close_to_wall:
                    break
                for penaltyy in range(-wall_penalty, wall_penalty + 1):
                    next_row = nextrow + penaltyx
                    next_col = nextcol + penaltyy
                    if not self.isValid(next_row, next_col):
                        too_close_to_wall = True
            if too_close_to_wall:
                continue

            nextcost = curcost + MOVE_COST[idx]
            if cost_map[nextrow][nextcol] == -1 or nextcost < cost_map[nextrow][
                nextcol]:
                cost_map[nextrow][nextcol] = nextcost
                heapq.heappush(points, (nextcost, nextrow, nextcol))

    np.savetxt("cost_map.txt", cost_map, fmt="%3d")

    highest_cost = 0
    targetrow, targetcol = -1, -1
    for row in range(num_rows):
        for col in range(num_cols):
            if highest_cost < cost_map[row][col]:
                highest_cost = cost_map[row][col]
                targetrow = row
                targetcol = col

    return (targetrow, targetcol)
```

E Multi-source Dijkstra Implementation with Cost Restriction

```
def find_target(self):
    num_rows, num_cols = self.cur_map.shape
    cost_map = [[-1 for _ in range(num_cols)] for _ in range(num_rows)]
    for x, y in self.visited_points:
        grid_x = round((x - self.map_origin.x) / self.map_res)
        grid_y = round(((y - self.map_origin.y) / self.map_res))
        convertx = self.map_width - 1 - grid_x
        converty = self.map_height - 1 - grid_y
        cost_map[convertx][converty] = 0
    points = []
    for x, y in self.visited_points:
        grid_x = round((x - self.map_origin.x) / self.map_res)
        grid_y = round(((y - self.map_origin.y) / self.map_res))
        convertx = self.map_width - 1 - grid_x
        converty = self.map_height - 1 - grid_y
        heapq.heappush(points, (0, convertx, converty))
    while points:
        curcost, currow, curcol = heapq.heappop(points)
        for idx in range(4):
            nextrow = currow + dr[idx]
            nextcol = curcol + dc[idx]

            if not self.isValid(nextrow, nextcol):
                continue

            too_close_to_wall = False
            for penaltyx in range(-wall_penalty, wall_penalty + 1):
                if too_close_to_wall:
                    break
                for penaltyy in range(-wall_penalty, wall_penalty + 1):
                    next_row = nextrow + penaltyx
                    next_col = nextcol + penaltyy
                    if not self.isValid(next_row, next_col):
                        too_close_to_wall = True
            if too_close_to_wall:
                continue

            nextcost = curcost + MOVE_COST[idx]
            if cost_map[nextrow][nextcol] == -1 or nextcost < cost_map[nextrow][
                nextcol]:
                cost_map[nextrow][nextcol] = nextcost
                heapq.heappush(points, (nextcost, nextrow, nextcol))

    np.savetxt("cost_map.txt", cost_map, fmt="%3d")

    highest_cost = 0
    targetrow, targetcol = -1, -1
    for row in range(num_rows):
        for col in range(num_cols):
            if highest_cost < cost_map[row][col] and cost_map[row][col] <=
                DISTANCE_THRESHOLD:
                highest_cost = cost_map[row][col]
                targetrow = row
                targetcol = col

    return (targetrow, targetcol)
```

F A* Search Algorithm Helper Function

```
def isValid(self, row, col):
    rowsize, colsize = self.cur_map.shape
    return 0 <= row < rowsize and 0 <= col < colsize and self.cur_map[row][col] <
        WALL_THRESHOLD and self.cur_map[row][col] != -1

def heuristic(self, curpoint, targetpoint):
    return (targetpoint[0] - curpoint[0]) ** 2 + (targetpoint[1] - curpoint[1]) **
         2

def reconstruct_path(self, parent_map, start, target):
    path = []
    node = target
    while node != start:
        path.append(node)
        node = parent_map[node]
    path.append(start)
    path.reverse()
    return path
```

G Plain A* Search Algorithm Implementation

```
def astar(self, target_row, target_col):
    start_row = self.currow
    start_col = self.curcol
    # Priority queue: (cost + heuristic, row, col)
    astar = []
    heapq.heappush(astar, (0, start_row, start_col))

    cost_map = { (start_row, start_col): 0 }
    parent_map = { (start_row, start_col): None }

    find_path = False
    while astar:
        cost, currow, curcol = heapq.heappop(astar)

        if (currow, curcol) == (target_row, target_col):
            find_path = True
            break

        for idx in range(4):
            nextrow = currow + dr[idx]
            nextcol = curcol + dc[idx]
            if not self.isValid(nextrow, nextcol):
                continue

            nextcost = cost + MOVE_COST[idx]
            if (nextrow, nextcol) not in cost_map or nextcost < cost_map[(nextrow,
                 nextcol)]:
                cost_map[(nextrow, nextcol)] = nextcost
                parent_map[(nextrow, nextcol)] = (currow, curcol)
                priority = nextcost + self.heuristic((nextrow, nextcol), (
                    target_row, target_col))
                heapq.heappush(astar, (priority, nextrow, nextcol))

    if find_path:
        print("A* Search found a path")
        path = self.reconstruct_path(parent_map, (start_row, start_col), (
            target_row, target_col))
        return path
    else:
        return []
```

H A* Search Algorithm with Wall Penalty Implementation

```
def astar(self, target_row, target_col):
    start_row = self.currow
    start_col = self.curcol
    # Priority queue: (cost + heuristic, row, col)
    astar = []
    heapq.heappush(astar, (0, start_row, start_col))

    cost_map = { (start_row, start_col): 0 }
    parent_map = { (start_row, start_col): None }

    find_path = False
    while astar:
        cost, currow, curcol = heapq.heappop(astar)

        if (currow, curcol) == (target_row, target_col):
            find_path = True
            break

        for idx in range(4):
            nextrow = currow + dr[idx]
            nextcol = curcol + dc[idx]
            if not self.isValid(nextrow, nextcol):
                continue

            cnt = 0
            for penaltyx in range(-wall_penalty, wall_penalty + 1):
                for penaltyy in range(-wall_penalty, wall_penalty + 1):
                    next_row = nextrow + penaltyx
                    next_col = nextcol + penaltyy
                    if not self.isValid(next_row, next_col):
                        cnt += 1

            nextcost = cost + MOVE_COST[idx] + cnt * 200
            if (nextrow, nextcol) not in cost_map or nextcost < cost_map[(nextrow,
                 nextcol)]:
                cost_map[(nextrow, nextcol)] = nextcost
                parent_map[(nextrow, nextcol)] = (currow, curcol)
                priority = nextcost + self.heuristic((nextrow, nextcol), (
                    target_row, target_col))
                heapq.heappush(astar, (priority, nextrow, nextcol))

    if find_path:
        print("A* Search found a path")
        path = self.reconstruct_path(parent_map, (start_row, start_col), (
            target_row, target_col))
        return path
    else:
        return []
```

I Point-to-point Movement Strategy Implementation

```
def move_through_path(self, points):
    # Debugging purposes
    for point in points:
        self.get_logger().info(f'Path: {point[0]}, {point[1]}')

    twist = Twist()
    for row, col in points:
        self.get_logger().info(f'Moving to {row}, {col}')
        while self.currow != row or self.curcol != col:
            # Triggering all callbacks
            for _ in range(3):
                rclpy.spin_once(self)
                time.sleep(0.1)

            self.get_logger().info(f'Current row col: {self.currow}, {self.curcol}
                ')
            self.get_logger().info(f'Target row col: {row}, {col}')
            self.get_logger().info(f'Current Angle: {self.initial_angle}')
            self.visited_points.add((self.currow, self.curcol))

            rotate_angle = 0
            speed = speedchange
            if self.curcol < col:
                rotate_angle = 360 - self.initial_angle
            elif self.curcol > col:
                rotate_angle = 360 - self.initial_angle
                speed = -speed
            elif self.currow < row:
                rotate_angle = 450 - self.initial_angle
            elif self.currow > row:
                rotate_angle = 450 - self.initial_angle
                speed = -speed

            self.get_logger().info(f'Rotate the bot ccw {rotate_angle} degree')
            if rotate_angle != 360:
                self.rotatebot(rotate_angle)
                self.initial_angle = (self.initial_angle + rotate_angle) % 360
            twist.linear.x = speed
            self.publisher_.publish(twist)
        self.stopbot()
```

J Path Clustering Implementation

```
def cluster_path(self, find_path):
    if find_path == []:
        self.get_logger().info('No path is found!')
        return
    clustered_path = []
    clustered_path.append(find_path[0])
    for idx in range(1, len(find_path)):
        point = find_path[idx]
        if self.point_to_point_distance(point, clustered_path[-1]) <=
            cluster_distance ** 2:
            continue
        clustered_path.append(point)
    return clustered_path
```

K Cluster-to-cluster Movement Strategy Helper Functions

```
def point_to_point_distance(self, a, b):
    return (a[0] - b[0]) ** 2 + (a[1] - b[1]) ** 2

def calculate_cw_rotation_angle(self, F, T):
    currow, curcol = F
    targetrow, targetcol = T
    coldiff = abs(curcol - targetcol)
    rowdiff = abs(currow - targetrow)
    if currow == targetrow and curcol == targetcol:
        return 0
    if currow > targetrow and curcol == targetcol:
        return 0
    if currow > targetrow and curcol < targetcol:
        return np.degrees(np.arctan(coldiff / rowdiff))
    if currow == targetrow and curcol < targetcol:
        return 90
    if currow < targetrow and curcol < targetcol:
        return 90 + np.degrees(np.arctan(rowdiff / coldiff))
    if currow < targetrow and curcol == targetcol:
        return 180
    if currow < targetrow and curcol > targetcol :
        return 180 + np.degrees(np.arctan(coldiff / rowdiff))
    if currow == targetrow and curcol > targetcol:
        return 270
    if currow > targetrow and curcol > targetcol:
        return 270 + np.degrees(np.arctan(rowdiff / coldiff))
```

L Cluster-to-cluster Movement Strategy Implementation

```
def move_through_path(self, points):
    if points == None:
        return
    twist = Twist()
    self.can_update = False
    self.get_logger().info(f'Path produced by A* Search')
    for row, col in points[1:]:
        self.get_logger().info(f'Row Col: {row}, {col}')
    please_redirect = False
    for row, col in points[1:]:
        if please_redirect:
            self.get_logger().info('Redirecting...')
            break
        self.get_logger().info(f'Moving to {row}, {col}')
        distance = self.point_to_point_distance((self.currow, self.curcol), (row,
            col))

        while distance > localization_tolerance ** 2:
            for _ in range(5):
                rclpy.spin_once(self)
                time.sleep(0.1)

            self.get_logger().info(f'Current row col: {self.currow}, {self.curcol}
                ')

            distance = self.point_to_point_distance((self.currow, self.curcol), (
                row, col))

            self.visited_points.append((self.cur_pos.x, self.cur_pos.y))
            while len(self.visited_points) > RESET_VISITED_POINTS_THRESHOLD:
                self.visited_points.pop(0)

            # Calculate the angle between (self.currow, self.curcol) and (row, col
                )
            map_angle = self.calculate_cw_rotation_angle((self.currow, self.curcol
                ), (row, col))
            map_angle = 360 - map_angle
            self.get_logger().info(f'Head up to the angle of {map_angle}')

            # Rotate the robot to that angle
            current_angle = math.degrees(self.yaw) + 180
            self.get_logger().info(f'Current Angle: {current_angle}')
            relative_angle = (current_angle - self.initial_angle + 360) % 360
            angle_to_rotate = (map_angle - relative_angle + 360) % 360
            self.rotatebot(angle_to_rotate)
            self.get_logger().info(f'Current Angle: {math.degrees(self.yaw)}')

            # Publish the twist
            twist.linear.x = speedchange
            self.publisher_.publish(twist)

        self.stopbot()
    self.can_update = True
    self.get_logger().info(f'Path traversed successfully')
```

M Callback for Updating Map (Initial Approach)

```
def occ_callback(self, msg):
    # find transform to obtain base_link coordinates in the map frame
    # lookup_transform(target_frame, source_frame, time)
    try:
        trans = self.tfBuffer.lookup_transform(
            'map', 'base_link',
            rclpy.time.Time(),
            timeout=rclpy.duration.Duration(seconds=0.1)
        )
    except (LookupException, ConnectivityException, ExtrapolationException) as e:
        print(e)
        return

    cur_pos = trans.transform.translation

    # get map resolution and map origin
    map_res = msg.info.resolution
    map_origin = msg.info.origin.position

    # get map grid positions for x, y position
    grid_x = round((cur_pos.x - map_origin.x) / map_res) # column in numpy
    grid_y = round(((cur_pos.y - map_origin.y) / map_res)) # row in numpy

    received_map = np.array(msg.data).reshape((msg.info.height, msg.info.width))
    adjusted_map = np.fliplr(np.rot90(received_map))

    self.currow = msg.info.width - 1 - grid_x
    self.curcol = msg.info.height - 1 - grid_y
    self.visited_points.add((self.currow, self.curcol))
    self.cur_map = adjusted_map
```

N Callback for Updating Map with Delay Flag

```
def occ_callback(self, msg):
    # find transform to obtain base_link coordinates in the map frame
    # lookup_transform(target_frame, source_frame, time)
    try:
        trans = self.tfBuffer.lookup_transform(
            'map', 'base_link',
            rclpy.time.Time(),
            timeout=rclpy.duration.Duration(seconds=1)
        )
    except (LookupException, ConnectivityException, ExtrapolationException) as e:
        print(e)
        return

    self.cur_pos = trans.transform.translation

    if self.can_update:
        # get map resolution and map origin
        self.map_res = msg.info.resolution
        self.map_origin = msg.info.origin.position
        self.map_width = msg.info.width
        self.map_height = msg.info.height
        received_map = np.array(msg.data).reshape((msg.info.height, msg.info.width
            ))
        adjusted_map = np.fliplr(np.rot90(received_map))
        self.cur_map = adjusted_map
        self.get_logger().info(f'Updating Map!')

    # get map grid positions for x, y position
    grid_x = round((self.cur_pos.x - self.map_origin.x) / self.map_res) # column
        in numpy
    grid_y = round(((self.cur_pos.y - self.map_origin.y) / self.map_res)) # row in
         numpy

    self.currow = self.map_width - 1 - grid_x
    self.curcol = self.map_height - 1 - grid_y
    self.visited_points.append((self.cur_pos.x, self.cur_pos.y))
    while len(self.visited_points) > RESET_VISITED_POINTS_THRESHOLD:
        self.visited_points.pop(0)
```

O Heat Interrupt Routine

```
# Calculate the shortest distance to the shooting areas
shortest_to_shooting_area = math.inf
for x, y in self.shooting_area:
    grid_x = round((x - self.map_origin.x) / self.map_res)
    grid_y = round(((y - self.map_origin.y) / self.map_res))
    convertx = self.map_width - 1 - grid_x
    converty = self.map_height - 1 - grid_y
    shortest_to_shooting_area = min(shortest_to_shooting_area, self.
        point_to_point_distance((convertx, converty), (row, col)))
print(f'Shortest distance to shooting area: {shortest_to_shooting_area}')

# If heat is detected and it is not one of the shooting area before, shoot
if self.heat_location != None and shortest_to_shooting_area >
    SHOOTING_AREA_THRESHOLD:
    while self.heat_location == 'right' or self.heat_location == 'left' or self.
        heat_location == 'forward':
        for _ in range(5):
            rclpy.spin_once(self)
            time.sleep(0.1)
        if self.heat_location == 'right':
            self.get_logger().info('Detecting heat in the right')
            self.stopbot()
            self.rotatebot(-HEAT_ROTATE_ANGLE)
        elif self.heat_location == 'left':
            self.get_logger().info('Detecting heat in the left')
            self.stopbot()
            self.rotatebot(HEAT_ROTATE_ANGLE)
        elif self.heat_location == 'forward':
            self.get_logger().info('Detecting heat in front')
            self.stopbot()
            twist.linear.x = speedchange
            self.publisher_.publish(twist)

    if self.heat_location == 'ok' and shortest_to_shooting_area >
        SHOOTING_AREA_THRESHOLD:
        self.get_logger().info('SHOOTTTTT!!! Stop for 9 seconds')
        self.stopbot()
        self.launch_ball_publisher.publish(String(data='ok'))
        self.shooting_area.add((self.cur_pos.x, self.cur_pos.y))
        please_redirect = True
        time.sleep(9)
        break
```