COVERAGE PATH PLANNING

SYED MAZHAR
u1988609

Supervisor: Tamara Petrovic
Co-supervisor: Felix

A Final Year Report submitted to School of Physical and Mathematical Sciences, Nanyang Technological University in partial fulfilment of the requirements for the Degree of

Intelligent Field Robotic Systems

June 2024

Table of Contents

0.1 Obstacle Avoidance

Obstacle avoidance is a crucial component of coverage path planning (CPP), applicable in various contexts such as agricultural fields, warehouses, and more. In agricultural fields, static obstacles include trees, rocks, houses, and other structures, collectively referred to as keep-out zones. Obstacles are represented as polygons of various shapes and sizes, which the robot must avoid and all the obstacle information is available beforehand.

One of the critical aspects of the CPP algorithm is its efficiency in obstacle avoidance. Given that the robot will encounter obstacles multiple times during its operation, it is essential that the algorithm minimizes the computation time required for each avoidance maneuver. The algorithm should be optimized to quickly navigate around obstacles. Path adjustments must be computed swiftly to ensure minimal disruption to the overall coverage path.

The obstacle avoidance algorithm in coverage path planning (CPP) consists of two main components:

1. Setup and dynamic grid generation.
2. Path validity checking and obstacle free path generation.

0.1.1 Setup and Dynamic Grid Generation

The initial setup begins once the algorithm receives all the data. The first crucial step is to account for the robot's physical dimensions by creating a configuration space. This ensures that the algorithm considers the robot's actual operating space, not just a point in the field.

CPP operates in continuous space for path planning, but representing obstacles solely in continuous space is computationally prohibitive, especially in obstacle-rich environments. On the other hand, completely using discrete space is impractical for large fields, such as agricultural areas, because methods like occupancy grids would be too expensive to generate and manage over extensive areas.

To address this, a hybrid approach is adopted. Continuous space is used for general path planning, while obstacles are represented in both continuous and discrete spaces. Discrete space involves creating separate occupancy grids for each obstacle. A critical challenge here is determining the size of the grid cells: they must be small enough to accurately represent the obstacle yet large enough to avoid excessive computational load. The algorithm addresses this by generating dynamic grids based on the size of the obstacle and the robot's non-holonomic constraints.

Dynamic Grid Generation: The initial objective is to dynamically generate an occupancy grid for each obstacle. The algorithm begins by extending the vertices of each polygonal obstacle

to create a safe zone around it. This extended obstacle is then approximated as a rectangle using the minimum and maximum coordinates of the extended polygon.

To generate the grid for this approximated obstacle, the algorithm considers the robot's non-holonomic constraints. It calculates a curve using the two extreme angles of the vision cone and a step length, ensuring the robot can navigate this curve with the minimum turning radius until a 90-degree turn is achieved. At this point, the changes in the x and y coordinates (dx and dy) are recorded. These values are used to determine the grid's dimensions, with an allowance of 1.5 times the curve distance along the robot's heading and half the obstacle's width. This setup ensures that the robot can navigate around the obstacle even with its non-holonomic constraints. The pseudocode for the dynamic grid generation for each obstacle is provided in (Algorithm 1)

Convention to be followed for the Obstacle Avoidance Algorithm:

- OB: Obstacle.
- OB_p: Polygonal obstacles.
- OB_e: Extended obstacles.
- g: grid.
- g_s: grids.
- p_{bu}: Buffer points.
- SP: Salient Point.
- P_{re}: Remaining path.

Notations for the Dynamic Grid Generation Algorithm:
- dx: Distance in the x direction at 90degree turn.
- dy: Distance in the y direction at 90degree turn.
- fd: Free grid space from the obstacle edge.

Algorithm 1 DynamicGridAlgorithm

Input: polygonal obstacles (OB_p), grid diameter (g_d), vision cone (VC), safe margin (sm)
Output: Extended polygonal obstacles (OB_e), grids (g_s), buffer points (p_{bu})

1: $dx, dy \leftarrow$ ComputeDistancesFor90DegreeTurn(VC)
2: $fd \leftarrow 1.5 \times dy$
3: $OB_e \leftarrow$ ExtendPolygonForSafeMargin(OB_p, sm)
4: $g_s \leftarrow$ ComputeGrid(OB_e, g_d, fd)
5: $p_{bu} \leftarrow$ PointsSurroundingObstacles(OB_e)
6: **return** OB_e, g_s, p_{bu}

Once the grid size is determined, the grid cells are generated. The cell size is chosen based on the robot's dimensions: larger robots do not require fine grids as they cannot move cell by cell. Therefore, an appropriate grid cell size is selected to suit the robot's size. For each obstacle, point data (centers of the complete grid) is generated over the occupancy grid. The algorithm extracts points around the obstacle's boundary, known as buffer points. These buffer points help in navigating around the obstacles efficiently.

0.1.2 Path validity checking and obstacle free path generation

This section outlines the second phase of the obstacle avoidance algorithm, focusing on the real-time aspects of navigation once the initial setup is complete. After setting up the obstacles and generating the dynamic grids, the algorithm proceeds with the regular behavioral approach for path planning. The pseudocode for the complete algorithm with obstacles is shown in (Algorithm 2)

Algorithm 2 CompleteBehavioralObstacles

Input: 2D points (p_{2d}), initial robot pose (r_{pos}), turning radius (R_{tu}), polygonal obstacles (OB_p), grid diameter (g_d), vision cone (VC)
Output: Dubins path P_{cd}

$OB_e, g_s, p_{bu} \leftarrow$ SetupObstacles(OB_p, g_d, VC, sm)
$p_{cl}, \delta_{\text{bc}} \leftarrow$ CentroidsAndAutoBehaviorShift(P_{2d}, R_{min}, R_{max}, N)
$p_r \leftarrow p_{cl}$
$\delta_c \leftarrow 0$
$P_c \leftarrow []$
while True **do**
    if $\delta_c < \delta_{\text{bc}}$ **then**
        $P_s, p_r \leftarrow$ Behavior_1(OB_e, g_s, p_{bu}, p_r, P_{cl}, N_s, r_{pos}, VC, C, R_{conc}, $step$)
    else
        $P_s, p_r \leftarrow$ Behavior_2(OB_e, g_s, p_{bu}, p_r, P_{cl}, N_s, r_{pos}, VC, C, R_{conc}, $step$)
    end if
    $P_c \mathrel{+}= P_s$
    $r_{\text{pos}} \leftarrow P_c[-1]$
    $\delta_c \leftarrow$ UpdateCoveragePerc(P_c, p_r)
    if len(P_s) $== 0$ **then**
        break
    end if
end while
$P_d \leftarrow$ DubinsPath(P_c, R_{tu})
$P_{re} \leftarrow$ PathAroundObstaclesAlgorithm(OB_e,P_r, r_{pos}, R_{tu})
$P_{cd} \leftarrow P_d + P_{re}$
return P_{cd}

The path generation process begins as described previously. However, this time, the algorithm includes a mechanism to detect path collisions with obstacles. If no obstacles are detected along the planned path, the behavioral algorithm operates normally. When an obstacle is detected, the algorithm initiates a sequence of steps to navigate around it. The pseudocode for the behavior

1 and behavior 2 with the inclusion of path validity check is shown in (Algorithm 3) and (Algorithm 4) respectively.

Description of the notations:
- P_{cu}: Current path
- P_{OBfree}: Obstacle free path

Algorithm 3 Behavior_1_with_Obstacles

Input: Extended polygonal obstacles (OB_e), grids (g_s), buffer points (p_{bu}), 2D points (P_{2d}), clustered points (p_{cl}), number of sample orientations (N_s), robot pose (r_{pos}), vision cone (VC), centroid (C), concurrent region radii (R_{conc}), step size (S)
Output: Complete straight path P_{cs}, remaining points p_r

$P_{cs} \leftarrow []$
$P_t \leftarrow [[] \text{ for } \_ \text{ in range}(N_s)]$
$p_{co} \leftarrow [[] \text{ for } \_ \text{ in range}(N_s)]$
$O_{sa} \leftarrow \text{SampleTheOrientations}(R_{pos}[2], N_s, S)$
for i, O **in** O_{sa} **do**
    $R_{pos}[2] \leftarrow O$
    while no point is visible **do**
        $p_v \leftarrow \text{ComputeVisionConePoints}(r_{pos}, VC)$
        $p_{po} \leftarrow \text{FindPotentialPoint}(r_{pos}, p_v)$
        if p_{po} is None **then**
            break
        end if
        $O_n \leftarrow \text{FromCurrentPoseToPotentialPoint}(r_{pos}, p_{po})$
        $P_{cu} \leftarrow [[r_{pos}, p_{po}]]$
        $P_{OBfree} \leftarrow \text{ComputeObstacleFreePath}(P_{cu}, O_n)$
        $P_t[i] \mathrel{+}= P_{OBfree}$
        $p_{int} \leftarrow \text{CheckIntermediatePoints}(p_r)$
        $P_t[i].\text{append}(p_{int})$
        $p_c[i].\text{append}(\text{len}(P_t))$
        $p_r \leftarrow p_{cl} - P_t$
        $r_{pos} \leftarrow P_{OBfree}[-1]$
    end while
end for
$O_{bi} \leftarrow \text{argmax}(p_c[:])$
$P_{cs} \leftarrow P_t[O_{bi}]$
$r_{pos} \leftarrow P_{cs}[-1]$
$p_{tu} \leftarrow \text{PotentialPointToTurn}(p_r, R_{\text{conc}}, r_{pos})$
$O_b \leftarrow \text{TowardsCentroid}(p_{tu}, C)$
$P_{cu} \leftarrow [[r_{pos}, p_{tu}]]$
$P_{OBfree} \leftarrow \text{ComputeObstacleFreePath}(OB_e, g_s, p_{bu}, P_{cs}, P_{cu}, O_b)$
$P_{cs} \mathrel{+}= P_{OBfree}$
$p_r.\text{remove}([P_{OBfree}])$
return P_{cs}, p_r

Algorithm 4 Behavioral_2_with_Obstacles

Input: Extended polygonal obstacles (OB_e), grids (g_s), buffer points (p_{bu}), 2D points (P_{2d}), clustered points (p_{cl}), number of sample orientations (N_s), robot pose (r_{pos}), vision cone (VC), centroid (C), concurrent region radii (R_{conc}), step size (S)
Output: Complete straight path P_{cs}, remaining points p_r

$P_{cs} \leftarrow []$
$P_t \leftarrow [[] \text{ for } \_ \text{ in range}(N_s)]$
$p_{co} \leftarrow [[] \text{ for } \_ \text{ in range}(N_s)]$
$O_{sa} \leftarrow \text{SampleTheOrientations}(R_{pos}[2], N_s, S)$
for i, O **in** O_{sa} **do**
    $R_{pos}[2] \leftarrow O$
    while no point is visible **do**
        $p_v \leftarrow \text{ComputeVisionConePoints}(r_{pos}, VC)$
        $p_{po} \leftarrow \text{FindPotentialPoint}(r_{pos}, p_v)$
        if p_{po} is None **then**
            break
        end if
        $O_n \leftarrow \text{FromCurrentPoseToPotentialPoint}(r_{pos}, p_{po})$
        $P_{cu} \leftarrow [[r_{pos}, p_{po}]]$
        $P_{OBfree} \leftarrow \text{ComputeObstacleFreePath}(P_{cu}, O_n)$
        $P_t[i] \mathrel{+}= P_{OBfree}$
        $p_{int} \leftarrow \text{CheckIntermediatePoints}(p_r)$
        $P_t[i].\text{append}(p_{int})$
        $p_c[i].\text{append}(\text{len}(P_t))$
        $p_r \leftarrow p_{cl} - P_t$
        $r_{pos} \leftarrow P_{OBfree}[-1]$
    end while
end for
$O_{bi} \leftarrow \text{argmax}(p_c[:])$
$P_{cs} \leftarrow P_t[O_{bi}]$
$r_{pos} \leftarrow P_{cs}[-1]$
$p_{tu} \leftarrow \text{PotentialPointToTurnCCW}(p_r, R_{\text{conc}}, r_{pos})$
$O_b \leftarrow \text{VectorsTowardsNextCCWPoint}(p_{tu}, p_r)$
$P_{cu} \leftarrow [[r_{pos}, p_{tu}]]$
$P_{OBfree} \leftarrow \text{ComputeObstacleFreePath}(OB_e, g_s, p_{bu}, P_{cs}, P_{cu}, O_b)$
$P_{cs} \mathrel{+}= P_{OBfree}$
$p_r.\text{remove}([P_{OBfree}])$
return P_{cs}, p_r

Collision Detection and Salient Point Identification

Upon detecting an obstacle in the path, the algorithm first identifies a line along the robot's heading and checks for intersection points with the obstacle. At this stage, buffer points

previously defined around the obstacle become critical. The algorithm identifies the furthest buffer point along the robot's heading, which marks the end of the line. It then calculates the perpendicular distances from this line to all buffer points, selecting two points on either side of the line at the maximum distance. These points are designated as salient points, serving as key navigational targets to avoid the obstacle.

Once the salient points are determined, they become the goal points for the robot to bypass the obstacle. The robot's current position is the starting point for this avoidance maneuver. The algorithm then checks whether the robot is inside the grid. If the robot is outside the grid, the salient points are directly used as goal points, and the robot navigates towards them following its non-holonomic constraints. The grid is designed such that if the robot is at the edge, it can avoid the obstacle by reaching these extreme points directly.

Graph-Based Path Finding

If the robot is inside the grid, a more complex process ensures that the robot can reach the salient points while adhering to its motion constraints. A graph-based approach is utilized to find the shortest and feasible path from the robot's current position to the salient points using the grid cells.

Efficient and rapid graph generation is crucial, as this process must occur each time an obstacle is encountered. The algorithm employs two extreme angles of the vision cone and a step length to create the graph. Each iteration produces a new generation of grid cells. For instance, if the robot occupies one cell, the next generation, based on the two extreme angles and step length, will occupy two cells. Subsequent generations expand similarly, covering more cells until the goal point is included in the graph.

One challenge in this graph-based approach is balancing the step length. If the step length is too short, the graph requires many generations to reach the goal point, increasing computational time. Conversely, if the step length is too long, the graph may become sparse, risking the possibility of not adequately covering the goal point and potentially passing through it.

To ensure an efficient and adaptive approach to obstacle avoidance, the algorithm dynamically determines the step length for graph generation. By allowing the algorithm to decide the step length, it can find a near-optimal length that generates a sparse graph initially, gradually becoming denser as it approaches the goal point. This adaptive strategy optimizes computational efficiency while ensuring thorough coverage.

Automating the selection of step length involved an experimental analysis to understand its dependence on various parameters. Notably, the distance to the salient point emerged as a significant factor. By conducting experiments with different salient point positions and step lengths, a linear relationship between the distance and step length was identified. Consequently, the algorithm fits a line to this data at the outset, enabling it to automatically determine the

dynamic step length based on the distance to the salient point. This near-optimal step length encourages the algorithm to generate a sparse graph initially, gradually densifying it as it approaches the goal point. This approach significantly enhances computational efficiency.

Once the graph is generated, multiple paths from the robot's position to the salient point are available. The algorithm employs an A* search over the graph to find the shortest path. The intermediate points along this path serve as the route to avoid the obstacle and reach the goal point efficiently and swiftly. Subsequently, the regular behavior resumes for further coverage of the designated points.

If the generated graph does not include the goal point, indicating that the goal point is unreachable within the constraints, the algorithm retraces one step back in the path. It then repeats the process from the point where the obstacle was detected, ensuring that the robot can circumvent the obstacle and complete its coverage path planning seamlessly.

The pseudocode for path validity check and finding obstacle free path is shown in (Algorithm 5) .

Description of the notations:

- obs_idx: obstacle index.
- S_l: Graph step length.
- gen_{max}: Maximum generation for graph.
- P_{sh}: Shortest path.

Algorithm 5 ComputeObstacleFreePath

Input: Extended obstacles (OB_e), grids (g_s), buffer points (p_{bu}), complete path (P_c), current path (P_{cu}), current orientation (O_{cu})
Output: Obstacle-free path (P_{OBfree})

1: $r_{pos} \leftarrow P_{cu}[-1]$
2: $is\_path\_in\_obstacle, obs\_idx \leftarrow$ CheckPath(P_{cu}, OB_e)
3: **if** $is\_path\_in\_obstacle$ **then**
4: $OB \leftarrow OB_e[obs\_idx]$
5: $g \leftarrow g_s[obs\_idx]$
6: $SP \leftarrow$ ExtractSalientPoints(OB, g, p_{bu}, r_{pos})
7: G.initialize()
8: $S_l \leftarrow$ AutoSelectStepLength(r_{pos}, SP)
9: G, $goal\_SP$, $is\_goal\_found \leftarrow$ GenerateGraph(G,r_{pos},SP,S_l,gen_{max})
10: **if** is_goal_found **then**
11: $P_{sh} \leftarrow$ AStarSearch(G,r_{pos}, $goal\_SP$)
12: $P_{OBfree} \leftarrow P_{sh}$
13: **return** P_{OBfree}
14: **else**
15: $r_{pos} \leftarrow P_c[-1]$ ▷ Move one step back in the path
16: **goto** GenerateGraph (step 9)
17: **end if**
18: **else**
19: **return** $[[P_{cu}[-1], O_{cu}]]$
20: **end if**

This iterative process repeats each time an obstacle is encountered, enabling the robot to efficiently navigate around obstacles and reach its designated goal points. This comprehensive approach ensures robust obstacle avoidance within the coverage path planning algorithm, facilitating efficient and timely completion of tasks.

One last change in the complete coverage with obstacles as compared to the regular coverage is replacing the DOTSP algorithm with another algorithm that can compute path for the remaining points left near the obstacles. Since, after the behavior 2, in this case there will be only some points left that will be close to the obstacles. This happens if the points are quite close to the obstacles. Path through this points cannot be computed through DOTSP algorithm as it collides with the obstacle and path validity is not available with DOTSP. Hence, a new algorithm is introduced to compute path through these points that are close to the obstacles.

This algorithm first associate each point to its nearest obstacle. Then, we will end up with each obstacles assciated with some points. Then, the algorithm first generates the visibility graph for each obstacle with points. Then, it will find the shortest path covering all the points of

first obstacle. Then, it will move to the next obstacle and cover all the points of that obstacle efficiently and so on. Eventually, it will cover all the remaining points close to all the obstacles. However, When dubins constraints are applied to this straight obstacle free path, some of the crves intersects with the corners of the obstacles due to looping of the path caused by the closeness of the points.

The pseudocode for this path finding for remaining points is shown in (Algorithm 6).
Notations:
- P_{list}: Points list
- P_{cu}: Current points
- P_{cov}: path through all points close to one obstacle.

Algorithm 6 PathAroundObstaclesAlgorithm

Input: Extended obstacles (OB_e), 2D points (P_{2d}), robot pose (r_{pos}), turning radius (R_{tu})
Output: Path (P)

$p_{list}, robot\_obs\_idx \leftarrow$ PointsCloseToObstacle(OB_e, P_{2d})
$p_{cu} \leftarrow p_{list}[robot\_obs\_idx]$ ▷ Robot close to an obstacle will be the first.
$OB_{cu} \leftarrow OB_e[robot\_obs\_idx]$
$P \leftarrow []$
for i **in** range(len(OB_e)) **do**
    G.initialize()
    $G \leftarrow$ GenerateVisibilityGraph(OB_{cu}, p_{cu}, r_{pos})
    $P_{cov} \leftarrow$ CoverageGraphSearch(G, r_{pos})
    $P \mathrel{+}= P_{cov}$
    $r_{pos} \leftarrow P_{cov}[-1]$
    $OB_{cu}, obs\_idx \leftarrow$ FindNearestObstacle(r_{pos}, p_{list})
    $p_{cu} \leftarrow p_{list}[obs\_idx]$
end for
return P

The obstacle avoidance approach in coverage path planning addresses several critical challenges with innovative solutions. These include dynamic grid selection to balance computational complexity and accuracy, employing a hybrid space approach to represent obstacles, and optimizing computational time through dynamic step length determination. The algorithm dynamically adjusts step length based on the distance to the salient point, ensuring near-optimal path planning efficiency. Additionally, utilizing both continuous and discrete spaces allows for efficient representation of obstacles. Moreover, employing an A* search algorithm facilitates the determination of the optimal shortest path, enabling swift and effective

obstacle avoidance. These integrated strategies ensure robust obstacle avoidance within the coverage path planning algorithm, enhancing overall efficiency and effectiveness.

The flowchart of the complete behavioal algorithm with obstacles can be visualized in the (Figure 1)

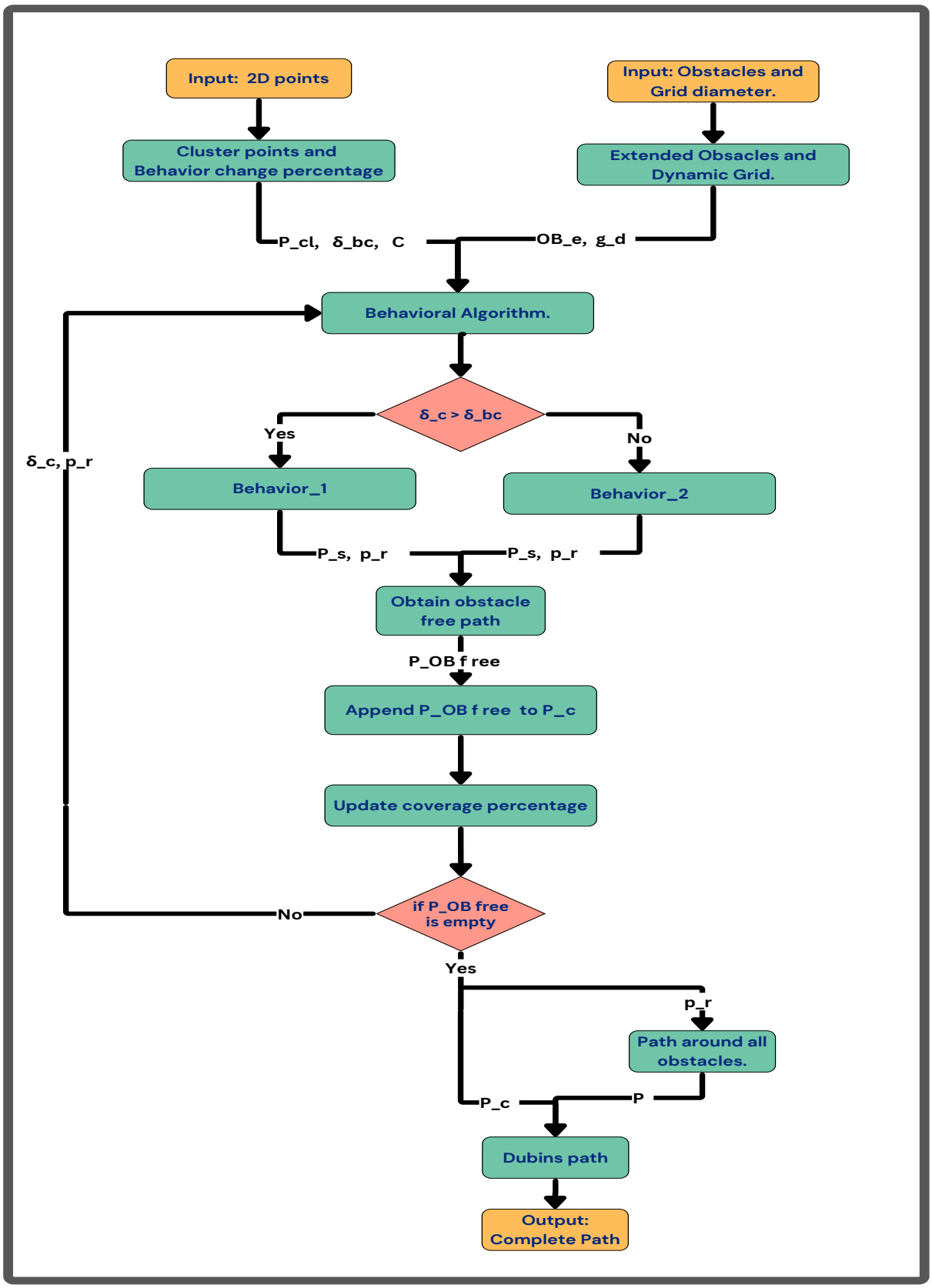

Figure 1: CCP Behavioral Algorithm with Obstacles flowchart.

1 Appendix

1.1 Experiment and Metrics Links

1. Effect of Behavior Change Percent: Click Here
2. Lawnmower Algorithm Performance Metrics: Click Here
3. Behavioral Algorithm Performance Metrics: Click Here
4. Obstacle Avoidance Algorithm Performance Metrics: Click Here

1.2 Videos

1. Simulation Videos link: Click Here
2. Real Robot Videos Link: Click Here