Documentation for LOSpathFinder Class

The LOSpathFinder class is designed to find a recovery path for a robot using Line of Sight (LOS) analysis. The class uses various computer vision and graph-based techniques to determine the best path based on the given map data.

Initialization

- **init(self, node)**
  - **Input:**
    - node (rclpy.node.Node): The ROS2 node.
  - **Output:**
    - None
  - **Description:**
    - Initializes the LOSpathFinder object. Sets up the ROS2 node, transforms, map subscription, and kernel matrices for noise reduction.

Callback Methods

- **map_callback(self, map)**
  - **Input:**
    - map (nav_msgs.msg.OccupancyGrid): The occupancy grid map.
  - **Output:**
    - None
  - **Description:**
    - Callback method for the map subscription. Updates the internal map data with the latest occupancy grid.

Transform Methods

- **get_transform(self, target_frame, source_frame)**
  - **Input:**
    - target_frame (str): The target frame for transformation.
    - source_frame (str): The source frame for transformation.
  - **Output:**
    - transform (geometry_msgs.msg.TransformStamped): The transformation between the target and source frames.
  - **Description:**
    - Retrieves the transformation between the target and source frames using the ROS2 TF2 buffer. Returns the

transformation if successful, otherwise logs a warning.

Pathfinding Methods

- **getRecoveryPath(self)**
  - **Input:**
    - None
  - **Output:**
    - keypoints (list of tuples): The key points for the recovery path.
  - **Description:**
    - Main method to find the recovery path. Processes the map to extract features, calculate LOS points, and determine the optimal path. Returns the key points for the recovery path.

Map Processing Methods

- **loadMap(self, map)**
  - **Input:**
    - map (nav_msgs.msg.OccupancyGrid): The occupancy grid map.
  - **Output:**
    - map_img (numpy.ndarray): The processed map image.
    - map_dimensions (tuple): The dimensions of the map image.
  - **Description:**
    - Loads the occupancy grid map, reshapes it, and applies scaling. Returns the processed map image along with its dimensions.
- **regionExtractor(self, map_img)**
  - **Input:**
    - map_img (numpy.ndarray): The map image.
  - **Output:**
    - regions (tuple): The extracted regions (freespace and boundaries).
  - **Description:**
    - Extracts different regions (freespace and boundaries) from the map image. Applies noise reduction techniques. Returns the extracted regions.

Line of Sight Methods

- **LOSpointCalc(self, img)**
    - **Input:**
        - img (numpy.ndarray): The image from which to calculate LOS points.
    - **Output:**
        - data_points (list of tuples): The calculated LOS data points.
    - **Description:**
        - Calculates the LOS points from the given image by thinning the image and finding contours. Returns the calculated data points.
- **calculateOptimalPoints(self, img, data_points)**
    - **Input:**
        - img (numpy.ndarray): The map image.
        - data_points (list of tuples): The LOS data points.
    - **Output:**
        - graph (dict): The graph of optimal points.
    - **Description:**
        - Calculates optimal points for pathfinding based on the data points. Returns a graph of optimal points.
- **filterGraph(self, img, list1)**
    - **Input:**
        - img (numpy.ndarray): The map image.
        - list1 (list of tuples): The list of points to filter.
    - **Output:**
        - best_lines (list of tuples): The filtered best lines.
    - **Description:**
        - Filters the graph of points to find the best lines. Returns the filtered best lines.
- **plotExtendedLine(self, img_display, img_gray, img_rgb, point1, point2, extension_length=1000)**
    - **Input:**
        - img_display (numpy.ndarray): The display image.
        - img_gray (numpy.ndarray): The grayscale image.
        - img_rgb (numpy.ndarray): The RGB image.
        - point1 (tuple): The first point.

        - point2 (tuple): The second point.
        - extension_length (int, optional): The length to extend the line (default is 1000).
    - **Output:**
        - clipped_line (tuple): The clipped extended line.
    - **Description:**
        - Plots an extended line on the image. Returns the clipped extended line.
- **clipExtendedLines(self, img, pnt1, pnt2, display)**
    - **Input:**
        - img (numpy.ndarray): The map image.
        - pnt1 (tuple): The first point.
        - pnt2 (tuple): The second point.
        - display (bool): Whether to display the result.
    - **Output:**
        - clipped_segment (tuple): The clipped line segment.
    - **Description:**
        - Clips the extended lines based on the boundaries of the map. Returns the clipped line segment.
- **findIntersectionPoints(self, line1, line2)**
    - **Input:**
        - line1 (tuple): The first line.
        - line2 (tuple): The second line.
    - **Output:**
        - intersection_point (tuple or None): The intersection point if found, otherwise None.
    - **Description:**
        - Finds the intersection points between two lines. Returns the intersection point if found.
- **calculateBestPoints(self, lines, img)**
    - **Input:**
        - lines (list of tuples): The list of lines.
        - img (numpy.ndarray): The map image.
    - **Output:**
        - LOS_points (list of tuples): The calculated LOS points.
    - **Description:**
        - Calculates the best points for pathfinding based on line intersections. Returns the calculated LOS points.

Utility Methods

- **calculateThreshold(self, line1, line2)**
    - **Input:**

        - line1 (tuple): The first line.
        - line2 (tuple): The second line.
    - **Output:**
        - threshold (float): The calculated threshold.
    - **Description:**
        - Calculates a threshold based on the distances between line segments. Returns the calculated threshold.
- **checkInRange(self, pnt1, pnt2, size)**
    - **Input:**
        - pnt1 (tuple): The first point.
        - pnt2 (tuple): The second point.
        - size (float): The size to check the range.
    - **Output:**
        - in_range (bool): Whether the points are within the specified range.
    - **Description:**
        - Checks if two points are within a specified range. Returns a boolean value.
- **mergeNearbyPoints(self, intersections)**
    - **Input:**
        - intersections (list of tuples): The list of intersection points.
    - **Output:**
        - None
    - **Description:**
        - Merges nearby points to reduce redundancy. Modifies the intersections list in place.
- **maximizeCoverage(self, LOS_points, threshold)**
    - **Input:**
        - LOS_points (list of tuples): The list of LOS points.
        - threshold (float): The threshold for coverage.
    - **Output:**
        - max_coverage_points (list of tuples): The points that maximize coverage.
    - **Description:**
        - Maximizes the coverage of LOS points based on a threshold. Returns the points that maximize coverage.
- **createGraph(self, LOS_points, robot_location, nearest_junc)**
    - **Input:**
        - LOS_points (list of tuples): The list of LOS points.
        - robot_location (tuple): The robot's location.

        - nearest_junc (list of tuples): The nearest junction points.
    - **Output:**
        - path (list of tuples): The calculated path.
    - **Description:**
        - Creates a graph from the LOS points and calculates the path using Dijkstra's algorithm. Returns the calculated path.
- **dijkstra(self, graph, start, end)**
    - **Input:**
        - graph (dict): The graph representation.
        - start (tuple): The start point.
        - end (tuple): The end point.
    - **Output:**
        - shortest_path (list of tuples): The shortest path.
    - **Description:**
        - Implements Dijkstra's algorithm to find the shortest path in a graph. Returns the shortest path.
- **visualizePath(self, text_list, img)**
    - **Input:**
        - text_list (list of tuples): The list of key points.
        - img (numpy.ndarray): The map image.
    - **Output:**
        - keypoints (list of tuples): The key points of the path.
    - **Description:**
        - Visualizes the calculated path on the image. Returns the key points of the path.
- **mouse_callback(self, event, x, y, flags, param)**
    - **Input:**
        - event (int): The mouse event.
        - x (int): The x-coordinate.
        - y (int): The y-coordinate.
        - flags (int): The flags.
        - param (any): The additional parameters.
    - **Output:**
        - None
    - **Description:**
        - Mouse callback for updating robot coordinates based on mouse movements.
- **robotNearestKeyPoint(self, gray_map, robot_location, LOS_points)**
    - **Input:**
        - gray_map (numpy.ndarray): The grayscale map.

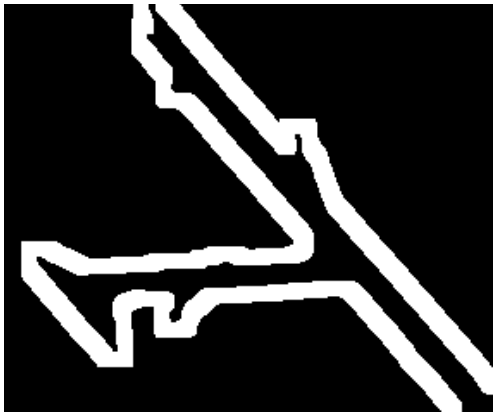
        - robot_location (tuple): The robot's location.
        - LOS_points (list of tuples): The list of LOS points.
    - **Output:**
        - nearest_junc (list of tuples): The nearest junction points.
    - **Description:**
        - Finds the nearest key points to the robot's location. Returns the nearest junction points.
- **scalingFactor(self, map_dimensions)**
    - **Input:**
        - map_dimensions (tuple): The dimensions of the map.
    - **Output:**
        - scaling_factor (float): The scaling factor.
    - **Description:**
        - Calculates the scaling factor based on map dimensions. Returns the scaling factor.
- **convertPixelsToMAPFrame(self, keypoints, mapOrigin, resolution)**
    - **Input:**
        - keypoints (list of tuples): The key points in pixel coordinates.
        - mapOrigin (tuple): The origin of the map.
        - resolution (float): The map resolution.
    - **Output:**
        - map_frame_points (list of tuples): The key points in map frame coordinates.
    - **Description:**
        - Converts key points from pixel coordinates to map frame coordinates. Returns the converted key points.

LOS Point Calculation Approach #01

Map processing
and
region extraction

Median filtered map

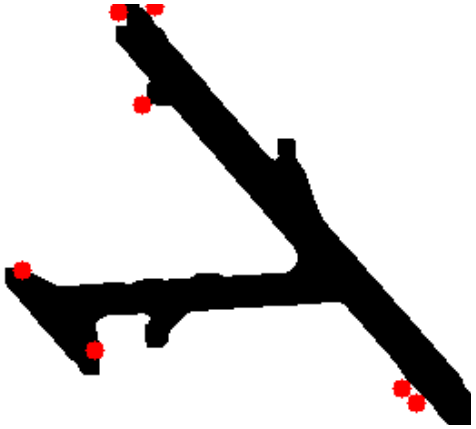
Thresholded binary map



Inflated boundary map

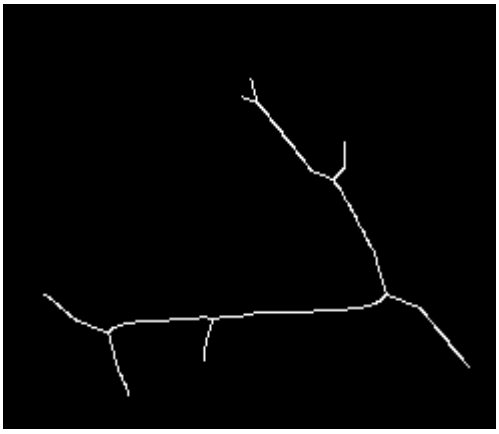
Frontier Detection

Frontier detection by
region extraction

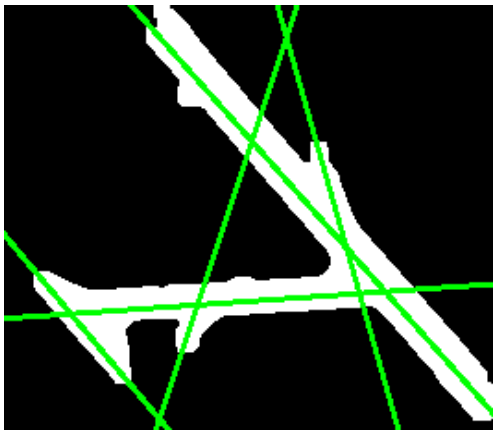


Detected center points
for frontiers

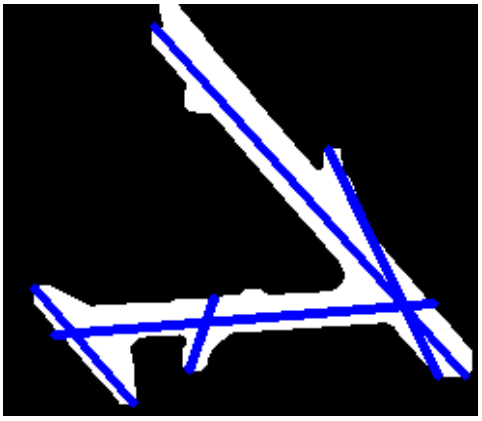
Map Skeletonization
and straight line estimation



Thinned map

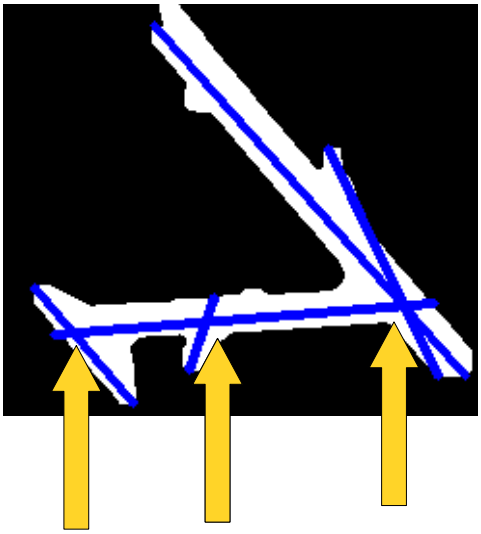


Straight line estimation
for thinned map

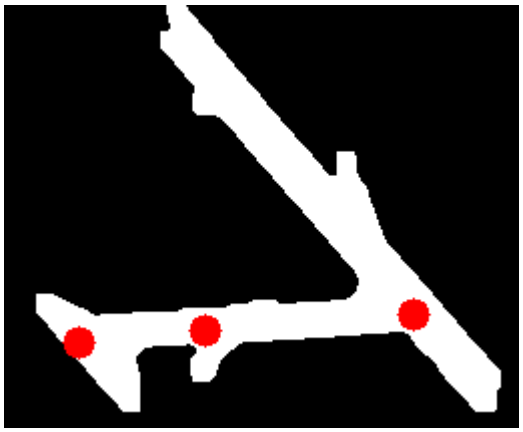


Clipping the lines such
that longest lines
represent each path

Calculating Junction points with maximum coverage

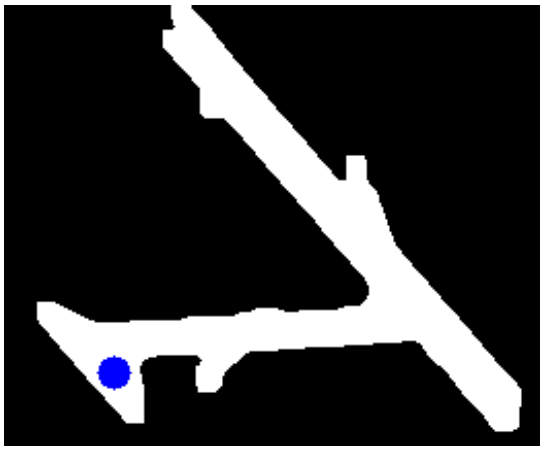


Best Coverage Points

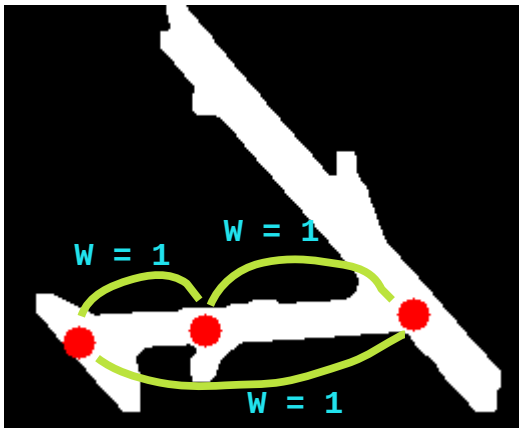


Detected Coverage Points

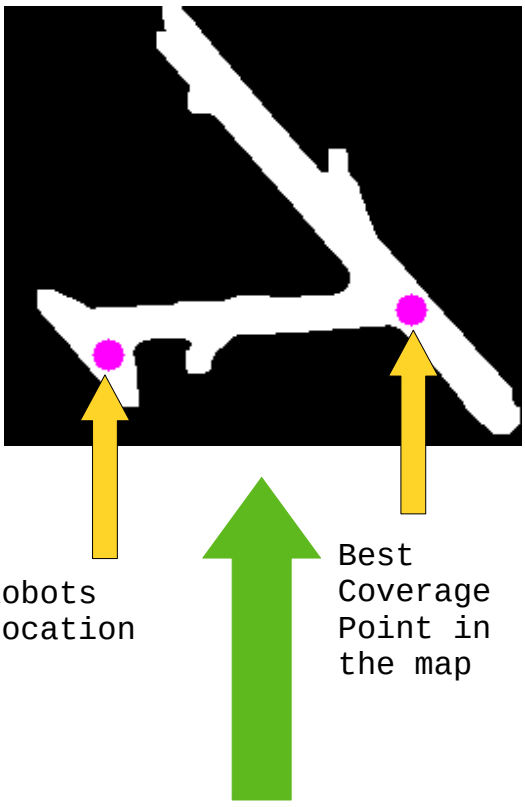
Get robots location and calculate MST



Robots location from map to baseframe transformation



MST calculation by Dijistras algorithm



Robots location

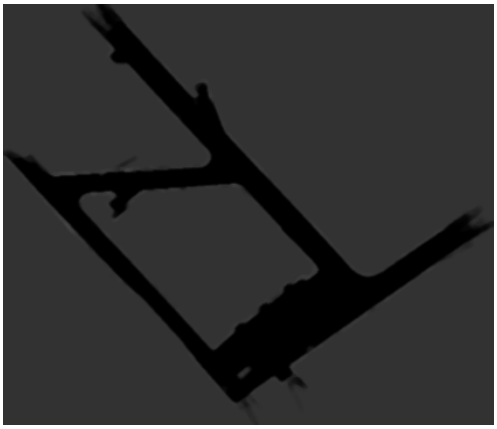
Best Coverage Point in the map

OUTPUT: Best coverage point of the map

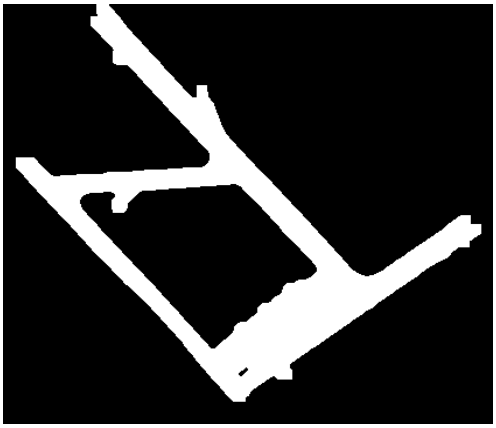
NOTE: In MST calculation, if a detected coverage point in a straight line with another coverage point, the weight between two points are assigned as 1.

LOS Point Calculation Approach #02

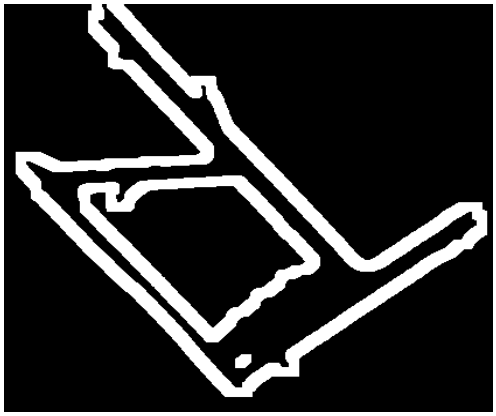
Map processing
and
region extraction



Median filtered map



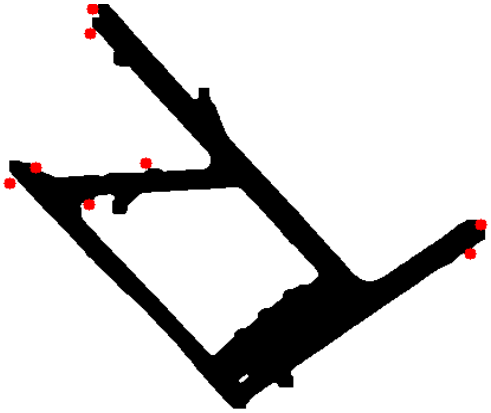
Thresholded binary map



Inflated boundary map

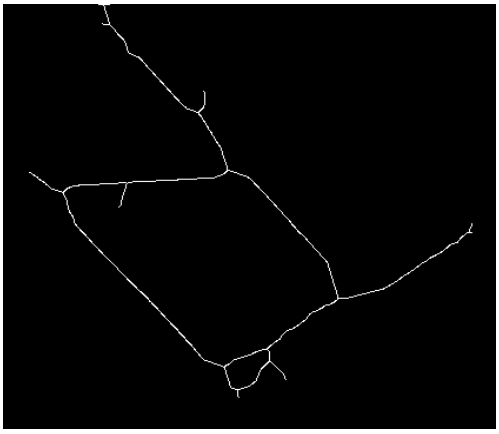
Frontier Detection

Frontier detection by
region extraction

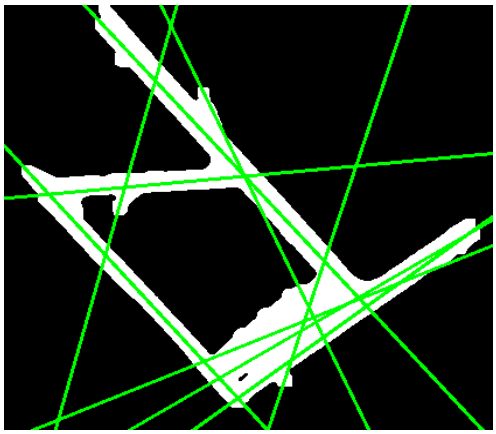


Detected center points
for frontiers

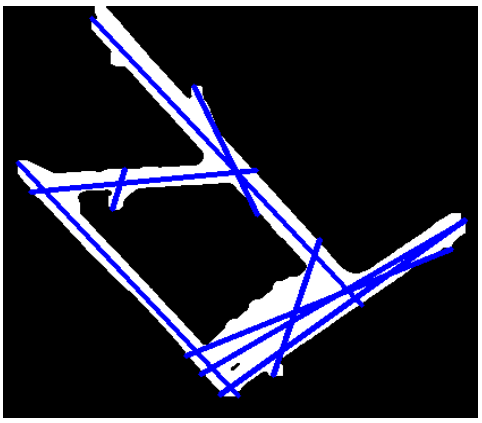
Map Skeletonization
and straight line estimation



Thinned map

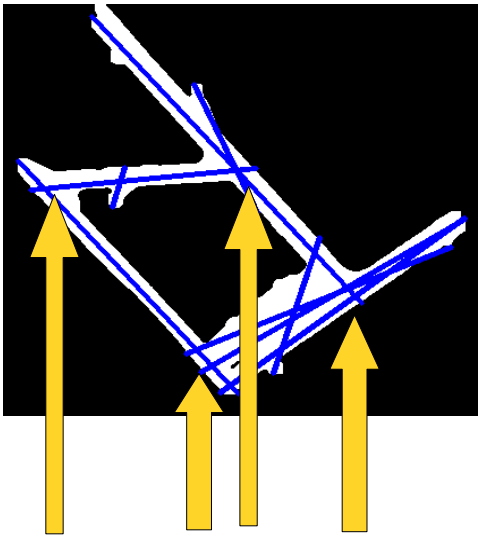


Straight line estimation
for thinned map

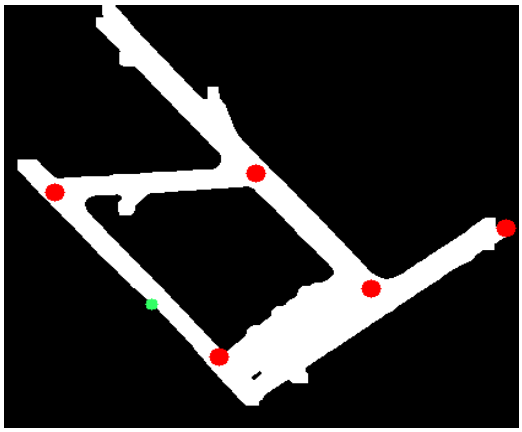


Clipping the lines such
that longest lines
represent each path

Calculating Junction points with maximum coverage

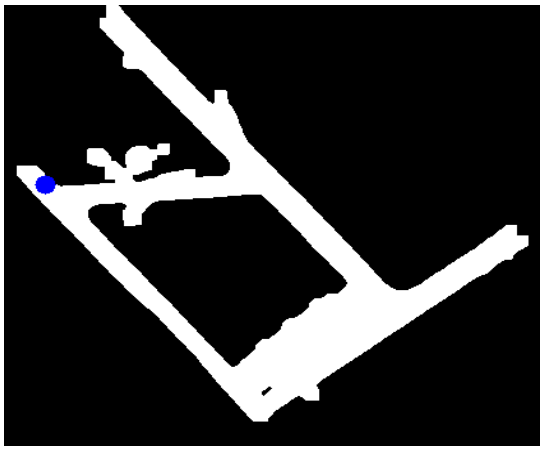


Best Coverage Points

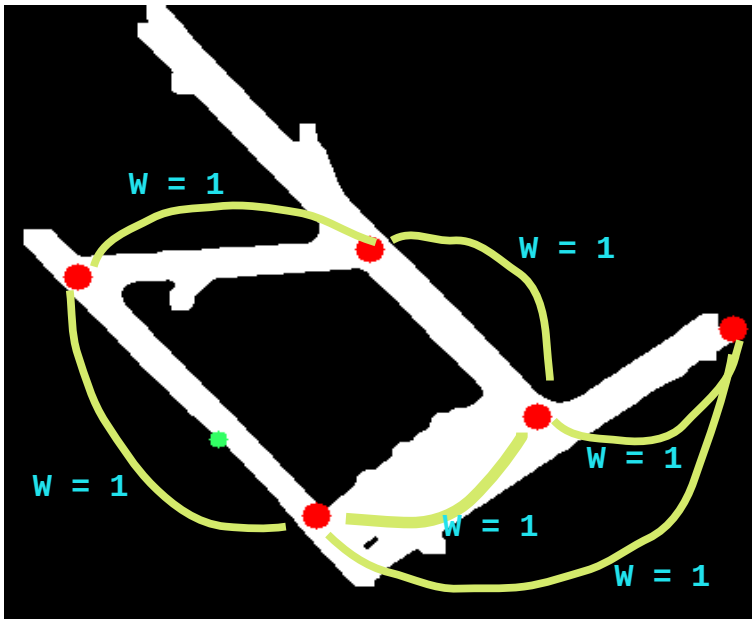


Detected Coverage Points

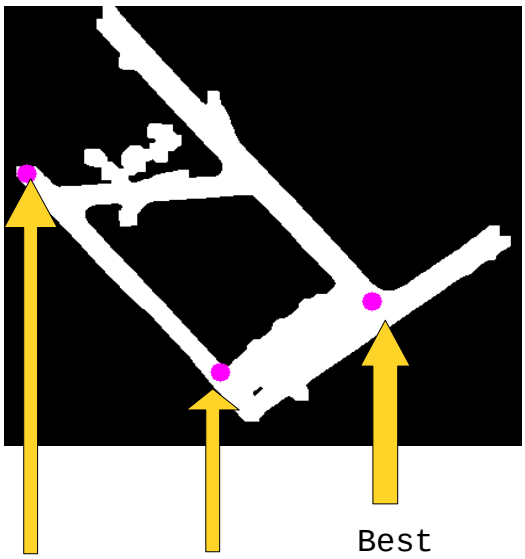
Get robots location and calculate MST



Robots location from map to baseframe transformation



MST calculation by Dijistras algorithm



Robots location

Path to best coverage point

Best Coverage Point in the map

OUTPUT: Best coverage point of the map