Special Implementation Report : Robotics and Spatial Intelligence

Part 1(A) Description of the Overall Approach:

System Overview

Initialization: The **GoalNavigationNode** class is initialized with a path to the world file. This class is responsible for controlling the robot's navigation.

Subscriptions and Publishers:

- The system subscribes to three ROS topics:
    - /map for receiving map data.
    - /goal_pose for receiving target destinations.
    - /current_position for the current position of the robot.
- It publishes to two topics:
    - /cmd_vel for commanding robot movement.
    - /path for visualizing the planned path.

Receiving Data:

- map_callback: Processes map data, extracting necessary information such as height and resolution.
- current_position_callback: Updates the robot's current position in the simulator node.
- goal_position_callback: Receives a new goal position and initiates the planning and execution process.

Path Planning:

- The PRM (Probabilistic Roadmap) algorithm is utilized for path planning.
- PRM takes the start and goal positions, a grid matrix representing the environment, and parameters like the number of samples and connection distance.
- It samples points in the environment, connects them based on proximity while avoiding obstacles, and then constructs a graph.
- The shortest path from the start to the goal is computed using Dijkstra's algorithm on this graph.

Special Implementation Report : Robotics and Spatial Intelligence

Executing Path:

- The execute_path method in **GoalNavigationNode** moves the robot along the computed path.
- It calculates the necessary linear and angular velocities to reach each intermediate goal in the path and publishes these velocities to /cmd_vel.

Handling Obstacles and Unreachable Goals:

- If the goal is in an unreachable location (like within an obstacle), the robot ignores the request.
- Obstacle detection and avoidance are handled during the path planning phase by the PRM algorithm.

Visualization and Monitoring:

- The planned path is published to the /path topic for visualization in rviz.
- The system constantly updates and responds to new goal poses, recalculating the path as needed.

The **execute_path** function in my code, which is responsible for moving the robot along the planned path, was reused from a previous project (Project 4b).

Part 1(B) The choices made in designing the program and the rationale behind those decisions:

When designing the navigation program for the differential drive robot, several key choices were made to ensure effective and safe navigation. Here's an overview of these choices and the rationale behind them:

Subscriptions to /map, /goal_pose, and /current_pos Topics:

Why: Subscribing to these topics allows the system to receive essential data for navigation: environmental layout (/map), destination points (/goal_pose), and the robot's current position (/current_position). This modularity in data handling makes the system adaptable to different environments and goals.

Special Implementation Report : Robotics and Spatial Intelligence

```
## creating the subscriptions to the relevant topics here
self.map_subscription = self.create_subscription(OccupancyGrid, '/map', self.map_callback, 10)
self.goal_subscription = self.create_subscription(PoseStamped, '/goal_pose', self.goal_position_callback, 10)
self.current_pose_subscriber = self.create_subscription(Pose, '/current_position', self.current_position_callback, 10)
```

Publishing to /cmd_vel and /path Topics:

Why: Publishing velocity commands to /cmd_vel is a standard way in ROS to control robot movement. Publishing the calculated path to /path helps in visualizing the robot's planned trajectory, which is useful for debugging and monitoring.

```
## Publishing to the /cmd_vel topic (for movement) and /path topic for visualization in Rviz
self.velocity_publisher = self.create_publisher(Twist, '/cmd_vel', 10)
self.path_publisher = self.create_publisher(Path, '/path', 10)
```

Handling Unreachable Goals:

Why: The system checks if a goal is reachable and ignores it if not. This decision avoids unnecessary computations and potential errors in trying to reach an impossible location.

Using a Graph-Based Approach for Path Planning:

Why: A graph-based approach simplifies the problem of navigating through an environment with obstacles. By representing the environment as a graph, standard pathfinding algorithms like Dijkstra's or A* can be used to find the shortest path (Dijkstra used in this case)

```
def build_graph(self,edge_list) -> nx.Graph:
    """
    Creates a graph based on connected points.
    """
    graph = nx.Graph()
    for connection in edge_list:
        graph.add_edge(connection[0], connection[1])
    return graph
```

Choosing the PRM Algorithm:

Special Implementation Report : Robotics and Spatial Intelligence

Why PRM: The PRM algorithm is an effective choice for path planning in known environments with a variety of obstacles. It is particularly suitable for high-dimensional spaces and complex environments where traditional grid-based algorithms might be inefficient. PRM works by randomly sampling points in the free space, connecting these points to form a network (or graph) of possible paths, and then finding the shortest path within this network. This randomness in sampling allows the algorithm to explore the environment thoroughly and is particularly effective in handling large, open spaces interspersed with obstacles.

```python
class PRM:
    def __init__(self, grid_matrix, start_point, goal_point, num_samples = 300, connect_distance = 2.0):
        """
        Initializes the Probabilistic Roadmap Planner
        """


        self.grid_matrix = grid_matrix

        self.start_coords = start_point
        self.goal_coords = goal_point
        self.sample_count = num_samples
        self.connection_threshold = connect_distance

        self.sampled_points = self.generate_samples()
```

Obstacle Detection: PRM incorporates an obstacle detection mechanism as part of its process to create the roadmap. Before adding a sampled point to the roadmap, the algorithm checks whether the point is in a collision-free space. This is done by evaluating the point and its surrounding area against the known map of the environment, ensuring that the roadmap only contains feasible paths.

```python
def is_obstacle_collision(self, point):
    """
    To check if a point, along with its adjacent points,
    is free from collision in the occupancy grid
    """

    x_coord, y_coord = point
    grid_height, grid_width = self.grid_matrix.shape

    # Define the points to check around the given point (including the point itself)
    points_to_check = [
        (x_coord, y_coord),  # Center point
        (min(x_coord + 1, grid_width - 1), y_coord),  # Right
        (max(x_coord - 1, 0), y_coord),  # Left
        (x_coord, min(y_coord + 1, grid_height - 1)),  # Up
        (x_coord, max(y_coord - 1, 0)),  # Down
        (min(x_coord + 1, grid_width - 1), min(y_coord + 1, grid_height - 1)),  # Diagonal top right
        (max(x_coord - 1, 0), min(y_coord + 1, grid_height - 1)),  # Diagonal top left
        (min(x_coord + 1, grid_width - 1), max(y_coord - 1, 0)),  # Diagonal bottom right
        (max(x_coord - 1, 0), max(y_coord - 1, 0))  # Diagonal bottom left
    ]

    # Check if any of the points are in collision
    return any(self.grid_matrix[int(check_y)][int(check_x)] == 1 for check_x, check_y in points_to_check)
```

Planning Path Between Obstacles:

Why This Approach: When connecting the sampled points, PRM considers whether a direct path between two points is feasible without intersecting any obstacles. This strategy effectively creates a network of collision-free paths, weaving through the obstacles. By considering local connections and avoiding collisions in this manner, PRM can efficiently chart paths through complex environments.

```python
def check_path_collision(self, start_index: int, end_index: int) -> bool:
    """
    Determines if the direct path between two points intersects with any obstacles.
    """
    start_point = self.sampled_points[start_index]
    end_point = self.sampled_points[end_index]

    start_x, start_y = start_point
    end_x, end_y = end_point
    step_range = 40

    for step in range(step_range):
        interpolation_factor = step / step_range
        interpolated_x = start_x * interpolation_factor + end_x * (1 - interpolation_factor)
        interpolated_y = start_y * interpolation_factor + end_y * (1 - interpolation_factor)

        if self.is_obstacle_collision((interpolated_x, interpolated_y)):
            return True

    return False
```

Special Implementation Report : Robotics and Spatial Intelligence

Using Dijkstra's Algorithm for Pathfinding:

Why Dijkstra's Algorithm: Once the PRM has constructed a graph representing the navigable space, the next step is to find the shortest path from the start to the goal. Dijkstra's algorithm is a well-established method for finding the shortest path in a graph, particularly effective when dealing with non-negative weights, as is the case with distances in the PRM graph. It guarantees to find the shortest path in terms of the cumulative distance between points, which is a primary objective in path planning. The algorithm is both efficient and reliable, making it a popular choice for pathfinding in robotics.

```python
def find_shortest_path(self,navigation_graph) -> list:
    """
    Uses Dijkstra to find the shortest path.

    """
    start_index = np.argmin(np.linalg.norm(np.array(self.sampled_points) - np.array(self.start_coords), axis=1))
    goal_index = np.argmin(np.linalg.norm(np.array(self.sampled_points) - np.array(self.goal_coords), axis=1))

    shortest_path = nx.shortest_path(navigation_graph, start_index, goal_index)

    path_coordinates = [self.sampled_points[idx] for idx in shortest_path]

    return path_coordinates
```

Part 1(B): Did the results meet my expectations?

Yes, The results of the project met my expectations primarily because of the successful implementation and integration of the Probabilistic Roadmap (PRM) algorithm within a ROS-based environment, effectively utilizing Rviz for interactive goal setting. The PRM algorithm is renowned for its proficiency in navigating complex environments, and its successful application in this project indicates a robust and efficient path planning strategy. This is evident from the algorithm's ability to accurately navigate to goal positions set through Rviz's 2D Goal Pose feature, demonstrating its effectiveness in real-world scenarios. Additionally, the system's capability to dynamically adjust to new goal poses and avoid obstacles aligns with essential requirements for autonomous navigation systems. The successful execution of these functionalities showcases a well-structured and thoughtfully designed system that also affirms the practical applicability of the theoretical principles underpinning the PRM algorithm.

Special Implementation Report : Robotics and Spatial Intelligence