**Autonomous Navigation Robot with Pillar Avoidance**

**Introduction**

This Python script implements an autonomous navigation algorithm for a robot using the Pygame library. The robot is tasked with navigating from its starting position to the center of a rectangular pillar while avoiding collisions with the pillar.

**Dependencies**

* Python 3.10
* Pygame library

**Features**

* Automatic calculation of the center of the rectangular pillar.
* Navigation algorithm to guide the robot to the center of the pillar while avoiding collisions.
* Visualization of the robot's movement and the pillar on the Pygame window.

**Usage**

1. Install Python and Pygame if not already installed.
2. Run the Python script using the command **python autonomous\_navigation\_robot.py**.
3. The Pygame window will display the robot's movement towards the center of the pillar while avoiding collisions.

**Code Structure**

* **Initialization**: Pygame is initialized, and screen dimensions are set.
* **Constants**: Definitions of colors and frame rate.
* **Pillar Coordinates**: Coordinates of the rectangular pillar are defined.
* **Center Calculation**: Function to calculate the center of the pillar.
* **Robot Position**: Starting position of the robot.
* **Navigation Algorithm**: Function to navigate the robot to the center of the pillar while avoiding collisions.
* **Draw Functions**: Functions to draw the pillar and the robot on the Pygame window.
* **Path Generation**: Path calculated by the navigation algorithm.
* **Main Loop**: Pygame main loop to continuously update the display.

**Pillar Avoidance Algorithm**

The navigation algorithm ensures that the robot avoids collisions with the pillar while moving towards the center. It navigates the robot horizontally and vertically, adjusting its path when encountering the pillar. Special cases are handled when the robot needs to navigate around the pillar without touching it.

**Steps involved in my algorithm:**

Basically from the initial point, it first moves along x axis to match the robot coordinate with x coordinate of the center and then moving along y axis to also match the y coordinate of the center point.

If any pillar comes on the way while moving along the x axis, The following cases are executed based on the pillar location:

* If the y coordinate of the pillar matches the top, then the robot tackles it by moving one step down along y axis and one step (right if the robot is moving along positive x axis or left if the robot is moving along negative x axis).
* If the y coordinate of the pillar matches the bottom, then the robot tackles it by moving one step up along y axis and one step (right if the robot is moving along positive x axis or left if the robot is moving along negative x axis).
* If the x coordinate of the pillar matches the left or right, then the robot tackles it by moving one half rotation around the pillar.

If any pillar comes on the way while moving along the y axis, then the pillar is tackled by moving one half rotation around the pillar to reach the inner region.

**Conclusion**

This script demonstrates a simple yet effective algorithm for autonomous navigation in the presence of obstacles. It can serve as a foundation for more complex navigation systems in robotics applications.

Github link : [Katomaran\_task/Robot\_center\_movement at master · hit1403/Katomaran\_task (github.com)](https://github.com/hit1403/Katomaran_task/tree/master/Robot_center_movement)

**Robot Navigation Simulation Documentation**

**Introduction**

This documentation outlines the functionality and implementation of a robot navigation simulation within a 10x10 grid. The simulation includes creating a grid with random obstacles, accepting user input for start and end points, calculating the shortest path using the A\* algorithm, and visualizing the grid, obstacles, and path using Pygame.

**Grid and Obstacles**

A 10x10 grid is created where each cell can either be empty or contain an obstacle. The grid is represented as a 2D list, and obstacles are randomly placed in 30 cells at the start of the simulation. Obstacles are represented by the value 1 in the grid.

**Visualization**

The **draw\_grid** function is responsible for drawing the grid on the Pygame window. Each cell is drawn with a border, and coordinates are displayed within each cell. The function also highlights the start point, end point, and the path found by the A\* algorithm using different colors:

* White for empty cells.
* Black for obstacle cells.
* Green for the start point.
* Red for the end point.
* Blue for the cells along the path.

**User Input**

The simulation prompts the user to enter the start and end points within the grid. The input points are validated to ensure they are within bounds and not placed on obstacle cells. If the input is invalid, the user is asked to re-enter the points.

**A\* Pathfinding Algorithm**

The A\* algorithm is used to find the shortest path from the start point to the end point while avoiding obstacles. The algorithm uses a heuristic function to estimate the cost from the current cell to the end point. The open set is managed using a priority queue to explore the most promising paths first. The algorithm keeps track of the path using a dictionary that maps each cell to its predecessor.

**Main Simulation Loop**

The main function of the simulation consists of several key steps:

1. Initial visualization of the grid with obstacles.
2. Getting user input for the start and end points.
3. Performing A\* pathfinding to compute the shortest path.
4. Main loop for visualizing the grid with the computed path.

**Initial Visualization**

The initial grid with obstacles is displayed to the user using the **draw\_grid** function. This provides a visual reference of the grid layout before inputting the start and end points.

**Getting User Input**

The user is prompted to enter the start and end points. These points are validated to ensure they are within the grid and not placed on obstacles.

**Pathfinding**

Once valid start and end points are provided, the A\* algorithm calculates the shortest path. If a path is found, it is displayed on the grid. If no path is found, the user is informed.

**Main Loop for Visualization**

The main loop continuously updates the Pygame window to visualize the grid, obstacles, start point, end point, and the path. The loop listens for Pygame events, such as quitting the window, to ensure proper handling of user interactions.

After visualizing the path, the user is prompted to continue with new inputs or exit the simulation. If the user chooses to continue, the process repeats; otherwise, the simulation ends.

**Conclusion**

The robot navigation simulation provides a visual and interactive way to understand grid-based pathfinding algorithms. By using Pygame for visualization and the A\* algorithm for pathfinding, the simulation demonstrates how a robot can navigate a grid with obstacles efficiently.

Github link: [Katomaran\_task/Grid\_Robotics at master · hit1403/Katomaran\_task (github.com)](https://github.com/hit1403/Katomaran_task/tree/master/Grid_Robotics)