NAME: Chidera L. Idabor

CLASS: Algorithmic Robotics

BUG 0 IMPLEMENTATION REPORT

**ALGORITHM DESCRIPTION**

The Bug 0 algorithm is a simple and greedy approach in robot motion planning. The core idea is for the robot to move directly towards the goal in a straight line unless it encounters an obstacle. Once the robot hits an obstacle, it follows the boundary of the obstacle until it can resume its direct path toward the goal. Unlike more advanced Bug algorithms (such as Bug 1 or Bug 2), Bug 0 does not retrace its steps or seek to minimize the path length. It simply follows the obstacle’s edge until it finds an opening to move toward the goal again.

**IMPLEMENTATION APPROACH**

In my implementation of Bug 0, I broke down the problem into several key components:

1. Grid Creation:

I modeled the environment as a 2D grid where free space is represented by 0 and obstacles by 1. I created a function that could generate different types of obstacles (rectangles, L-shapes, and circles), allowing me to test the algorithm against various shapes.

2. Direct Motion Toward the Goal:

The robot attempts to move in a straight line toward the goal by incrementally adjusting its position using the sign function, which moves it closer to the goal. This continues until either the robot reaches the goal, or it encounters an obstacle.

3. Boundary Following:

When the robot encounters an obstacle, it switches to boundary-following mode. In this mode, it moves along the obstacle’s boundary while avoiding revisiting previous boundary points. The goal is to resume its path toward the goal once the obstacle is cleared.

4. Path Recording and Visualization:

Throughout the algorithm, I recorded the robot's path, which allowed me to visualize the movement. I used `matplotlib` to display the grid, the obstacles, and the robot’s path, with the added feature of animating the movement step-by-step.

5. Shape-based Obstacle Creation:

I implemented a flexible system for obstacle creation that allows the environment to switch between different shapes (rectangle, L-shape, and circle). This enabled me to test the Bug 0 algorithm across various types of obstacles and scenarios.

**PARAMETERS USED**

1. Grid Size: The size of the grid, defined by rows and columns, sets the boundaries for the robot's movement.
2. Max Steps: This parameter limits the number of steps the robot can take, preventing it from getting stuck in an infinite loop.
3. Shape Argument: This allows me to dynamically define which obstacle shape is present in the environment (rectangle, L-shape, or circle).
4. Boundary Following Directions: The robot checks four directions (right, down, left, and up) when following the boundary of an obstacle.

**GITHUB FOR VERSION CONTROL**

Throughout the development process, I used GitHub to track my changes, maintain version control, and document my progress. This allowed me to easily revert to previous versions when needed and experiment with different approaches without losing track of the main implementation.

**WHY I CHOSE BUG 0**

Initially, I experimented with the Bug 1 and Bug 2 algorithms, which have a more difficult strategy for navigating obstacles. However, I found that these algorithms were more complex to implement, especially in environments with diverse obstacle types like rectangles, L-shapes, and circles. Given the challenges and my goal of implementing a reliable, easier-to-debug solution, I decided to use Bug 0. Its simplicity, while not the most efficient, made it a more practical choice for this project.

**CHALLENGES FACED**

1. Navigating Complex Shapes: One of the main challenges I encountered was getting the robot to properly navigate around obstacles with irregular shapes like rectangles and L-shapes. Circular obstacles were easier to handle because the boundary-following logic worked more naturally, but the more angular shapes posed difficulties for the Bug 1 and Bug 2 algorithms.
2. Rejoining the Path: In Bug 2, the robot is expected to rejoin the M-line (the direct line between the start and the goal) after circumventing an obstacle. Implementing this rejoining step proved tricky, as it required precise checks to determine the right time to rejoin the M-line, leading to issues where the robot would not rejoin the path early enough or would get stuck.
3. Creating Circular Obstacles: Another challenge was accurately creating circular obstacles in a grid-based environment. Since the grid uses discrete cells, creating a perfect circle is difficult. I had to approximate the circle using the equation of a circle, but this led to jagged edges rather than a smooth boundary. This made the robot’s behavior slightly unpredictable when following the boundary of the circle, as it sometimes failed to detect the full circumference smoothly.
4. Visualization and Debugging: Implementing animation using `matplotlib` was challenging, especially ensuring that each step of the robot’s movement was accurately reflected in the visualization. Having real-time feedback was essential to monitor the robot’s behavior and correct any mistakes.

**HOW I OVERCAME CHALLENGES**

1. Simplifying to Bug 0: After struggling with the complexities of Bug 1 and Bug 2, I simplified the approach by focusing on Bug 0. The simpler boundary-following logic, without worrying about rejoining the M-line or retracing, allowed me to get reliable results across various environments.
2. Flexibility in Obstacle Creation: I developed an adaptive function to create different obstacle shapes (rectangle, L-shape, and circle) within the environment. This flexibility made testing easier and allowed me to focus on the robot’s interaction with different obstacles.
3. Iterative Visualization: By visualizing each step of the robot's movement in real-time, I was able to observe where it was encountering issues and refine the algorithm accordingly. This iterative process helped me debug and improve the robot’s pathfinding behavior.

**WHAT I LEARNED**

1. Algorithm Trade-offs: I learned that more complex algorithms (like Bug 1 and Bug 2) offer greater efficiency and functionality but are much harder to implement and debug. Simpler algorithms, like Bug 0, while less efficient, provide reliability and ease of understanding, making them a good starting point for pathfinding tasks.
2. Importance of Visualization: Being able to visualize the robot’s movement was crucial in understanding its behavior. It allowed me to quickly identify problems, such as the robot getting stuck or failing to follow the boundary correctly. Without visualization, debugging would have been significantly more challenging.
3. Handling Circular Obstacles in a Grid: Working with a grid-based environment made it difficult to accurately represent circular obstacles. The approximation of circles using a grid can lead to irregular edges, which may affect the robot’s boundary-following logic. This taught me that while grid-based environments are simple to implement, they have limitations when representing continuous shapes like circles.
4. Modularity in Code: I realized the importance of writing modular code that can be adapted to different situations. By designing the environment creation function to take different shapes as arguments, I was able to test the algorithm in various scenarios without rewriting large parts of the code.