Mobile Robots - Localization

Matias Cinera
Computer Science Department, *University of South Florida*
Tampa, Florida, 33612, USA
cinera@usf.edu

I. Introduction

This document is the report of my 5th lab from my Spring 2023 class CDA-6626 Autonomous Robots, as a graduate student. The individuals who are overviewing this project are our professor Dr. Alfredo Weitzenfeld (Professor in the department of Computer Science) and Mr. Chance Hamilton (Teaching Assistant & Graduate Student)

II. Localization

This lab aims to localize the robot and map its surroundings using motor control, lidar sensors, and cameras. The provided parameters are a global reference frame, with a 4x4 grid of tiles size 10x10 inches and 4 landmarks at each grid corner (cylinders). The goal of each task is to visit all tiles at least once.

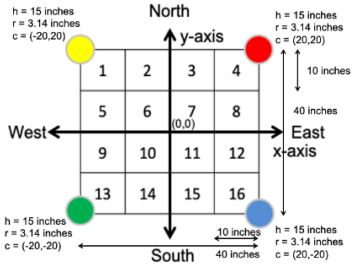

Figure 1: Global reference frame and grid cell numbering scheme

III. Motion Estimation

For Motion Estimation I assume the robot's starting position on the grid. Knowing this makes the task trivial, the only thing left to do is to develop a traversal algorithm.

A. Modeling the Robot and Grid

There There are a few variables to keep track of when solving this problem. The robot's current position (x, y), the current tile its in, and its current angle. Instead of building a graph based on the grid, I represented the grid as a 2d matrix, where each node can have at most 4 neighbors.

```
class RobotPose:
  def __init__(self, x, y, tile, theta):
    self.x = x
    self.y = y
    self.tile = tile
    self.theta = theta
```

Figure 2: Robot Class

```
grid = [[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]]
```

Figure 3: Grid Model, 0 = undiscoverred, 1= discovered

B. DFS Traversal

I coded a DFS-like traversal. First, I calculated the valid neighbors (edges) in real-time. Since Im not storing the neighbors of any tiles (edges) in a data structure, I have to check for valid neighbors in real time. A valid neighbor is undiscovered and has no wall between the current tile and the target tile. I also used a **stack** to keep track of the motions the robot is performing. In the case that the current tile has 0 valid neighbors, the robot will backtrack its motions (popping the stack) until the current tile has valid neighbors to traverse.

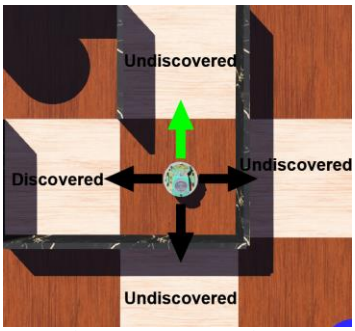

Figure 4: Illustration of algorithm: Checking for valid neighbors, in this case the robot only has 1 valid neighbor

IV. Measurement Estimation

A. Triliteration

Once 3 landmarks are found, calculating x & y with trilateration is trivial. Since the position of all landmarks is already known, I decided to always maintain a relatively certain order in which to process the landmarks (yellow, red, blue, green). Once x and y are calculated, traverse with dfs. After each waypoint, try to calculate again x and y.

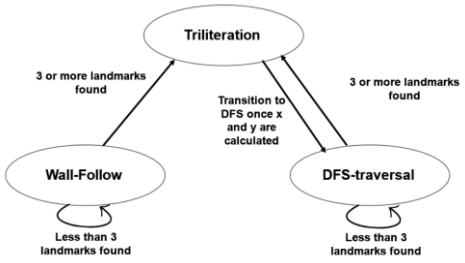

Figure 5: DFS + Triliteration algorithm.

V. Conclusions

The most difficult part of this project was implementing the data structures and logic of DFS traversal. Since that is the most important part of both algorithms. Trilateration can be a problem if passing the wrong arguments, so I implemented a helper function to make sure the order of the landmarks was correct.