University of California San Diego

CSE 276A: HW5
Roomba-like System

Orish Jindal, Varun Vupparige
December 6, 2022

1 Introduction

The adoption of robotics solutions for consumer use during the early 21st century marked the rise in the use of automatic vacuum cleaning robots. Like any other autonomous system today, the autonomous vacuum cleaner also called Roomba, a generic trademark, consists of various subsystems such as sensing, perception, path planning, and controls.

In this assignment, we aim to design a Roomba-like system capable of mapping its environment and providing a certain level of area coverage for the supposed cleaning action, similar to a Roomba robot. The next section will explain our design motivation and architecture, modules' functionalities, and results.

2 Environment Representation

The figure shown below represents the actual map that needs to be covered by our Roomba-like robot. It is a 10ft $\times$ 10ft square map with 3 AprilTags (landmarks) on each of the top and bottom sides (as per the representation in the image) and 2 AprilTags (landmarks) on each of the left and right sides of the square. This is because the robot will be facing either one of the top or bottom most of the time during the whole operation.

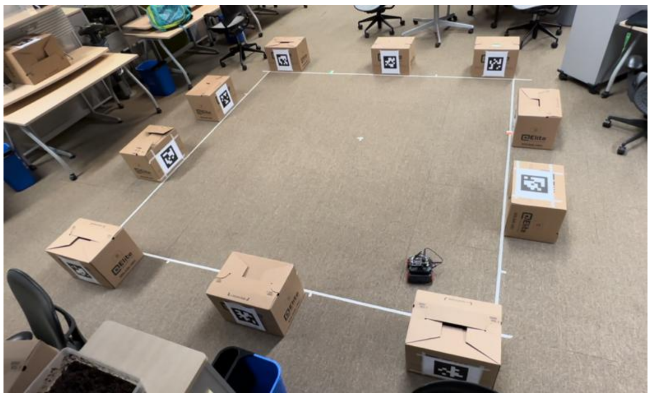

Figure 2.1: Ground Truth Environment with Landmarks

2.1 Representation of the map:

A grid map was prepared to represent the environment of 10ft $\times$ 10ft (3.048m $\times$ 3.048m), and the cleaner of the robot is roughly considered a square of 20cm $\times$ 20cm to be as close as possible to the robot's actual size. Now we divide the map into grids of size matching closely with the robot's cleaner to cover the maximum area with minimum runs. The calculation is as follows:

- Side of the square environment = 304.8 cm
- Side of the square cleaner = 20 cm
- Minimum number of grids required = 304.8/20 = 15.24

So the closest higher integer to 15.24 is 16. Hence, the map is divided into 16 $\times$ 16 grids of equal size by drawing 17 lines on each dimension (zero-indexed) with the yellow color representing the free space, which is shown below:

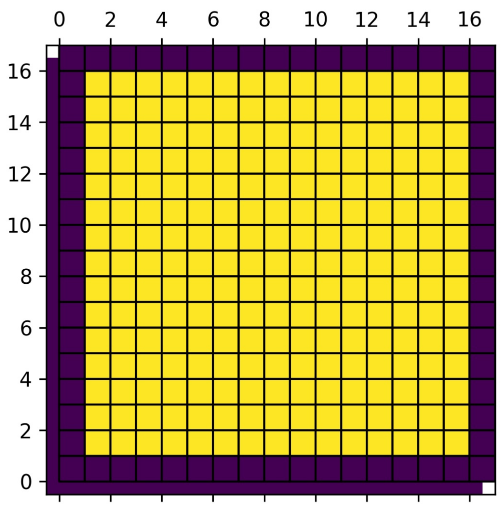

Figure 2.2: Grid map of mentioned size

3 Motivation and Architecture

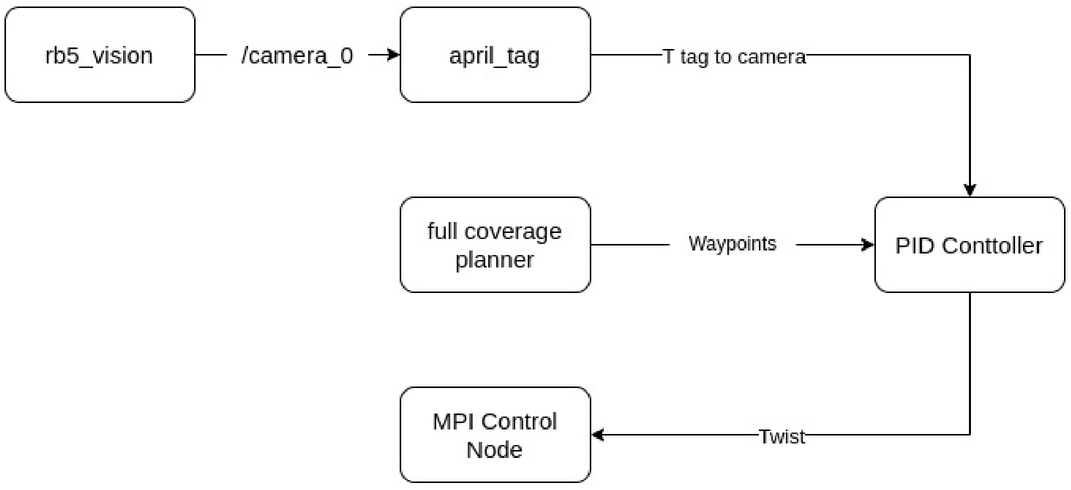

Figure 3.1: Control Flow/Architecture

The code architecture implemented to solve the problem statement includes the following four nodes. The function of each of the four nodes is described below:

- RB5 Vision: Camera node initialized using rb5 camera launch executable This node initializes the main camera node, published as camera 0. It enables the camera to capture image frames of the environment basis its field of view.
- April Tag: April tag detection node initialized using apriltag detection array executable. The April tag detection node subscribes to the camera node to identify April tags visible in the camera frame. Each AprilTag has been assigned a unique id. If the tags are present, this node calculates the position and orientation of the robot with respect to the tag in terms of (x, y, z) coordinates and orientation quaternion. This information is published to the /apriltag detection array rostopic.
- Path Planner: Python script computes the waypoints from start to goal state based on the Boustrophedon algorithm, and these waypoints are saved in a txt file to be read by the PID controller node. This algorithm is explained in detail in the next subsection.
- PID Controller: PID control loop is implemented which takes the waypoints calculated by the full coverage path planner and enables the robot to follow the desired trajectory using position feedback from the AprilTags.
  - This node subscribes to the April tag detection array rostopic to obtain the position and orientation of the robot concerning the tag.
  - As the world frame coordinates of the April tags are known, the world frame coordinates of the robot are calculated using the feedback from tags. This process is executed continuously to update the robot's current position at each step.
  - World frame coordinates of the robot are calculated by taking into account all the visible tags in the camera frame.
  - Handling of ambiguity: The algorithm is designed to consider the visibility of the repeated tags in the environment. While the case of unique tags is simple, the case of repeated tags is dealt with by including each set of world frame coordinates of the repeated tag. If the tag's id visible in the camera frame corresponds to that of a repeated tag, the world frame coordinates of the robot are calculated w.r.t. each instance of the repeated tag. For each set of robot coordinates, the error is calculated w.r.t. its previous position. The set of coordinates that give the least error is considered the new current position of the robot.
  - Once the current position is updated, the PID control loop calculates the error w.r.t. the target position and transforms it into a twist vector. This vector is published to the /twist rostopic.
  - These steps are executed continuously in a loop until all the waypoints required to follow the given trajectory are covered.
- MPI Control Node: MPI control node initialized using hw2_mpi_control_node executable. MPI control node subscribes to /twist rostopic to obtain twist vector calculated by the PID control loop. It is then scaled using a calibration factor. The Jacobian matrix transforms the twist vector into PWM signals for individual motors. The PWM signals for each of the four motors are sent to MegaPiController firmware to control the motors individually.

3.1 Boustrophedon Algorithm

Coverage path planning determines a path that guarantees that an agent will pass over every point in a given environment. This procedure allows for a variety of applications. Most cheap automatic robot vacuum cleaners today combine three or four different path-planning algorithms to get the expected coverage as fast as possible. The price point of these robot cleaners is heavily dependent on the type of sensors being used. Thus the only way to crack this market is to have an efficient and robust path-planning algorithm to support the cheap sensors.

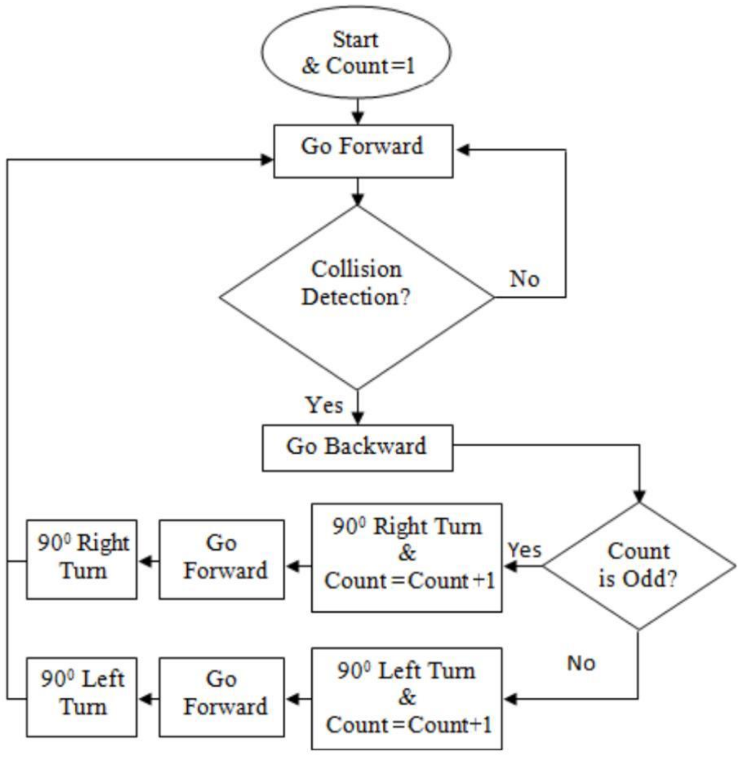

Figure 3.2: Flowchart of Boustrophedon Algorithm

We implemented the Boustrophedon Algorithm, which mimics an ox when plowing and makes the robot zig-zag motion from the opposite walls while traversing the room length-wise. When the algorithm is implemented, it checks if the robot is in the middle by checking if the rightmost grid space is occupied or not. If not, the robot moves to the right until it encounters a wall. Once it is adjacent to the walls, it starts going forward until it reaches the wall. Since our robot does not have a bumper/proximity sensor to know if it has reached the wall or not, this part is taken care of by our localization logic. Once it reaches the front-most grid cell, it turns 90 degrees counter-clockwise to the immediate left grid cell and turns 90 degrees counter-clockwise again. The robot cleaner then travels downwards till the bumper is triggered again. The robot cleaner will now turn 90 degrees left and then travels a distance equal to the robot cleaner's length and turns 90 degrees left again, making the robot cleaner face upwards. This is executed multiple times until the robot covers all the grid spaces. The flowchart explains the flow of operations in the Boustrophedon Algorithm.

4 Performance Guarantees

The algorithm implemented for this assignment's scope is capable of planning a path to cover 100% area of the environment when the obstacles are precisely the same size or a multiple of the size of grid blocks. Suppose this ideal case is not the reality. In that case, our algorithm will force the robot to consider grid blocks containing even a tiny part of a randomly shaped obstacle as occupied, thus dropping the coverage to less than 100%.

Typically use case of this robot is in environments that are filled with obstacles of varying shapes and sizes. To test the performance of our algorithm in cases with obstacles, we created a map with a single obstacle and two obstacles. Referring to fig 5.2 and 5.3, we can see that the algorithm is capable of full area coverage with multiple obstacles.

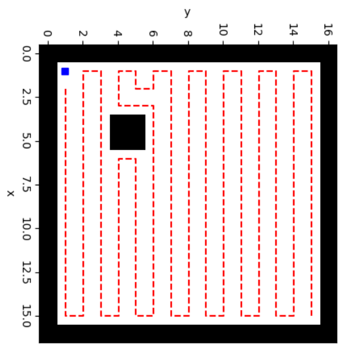

Figure 4.1: Path traced by Algorithm: Single Obstacle

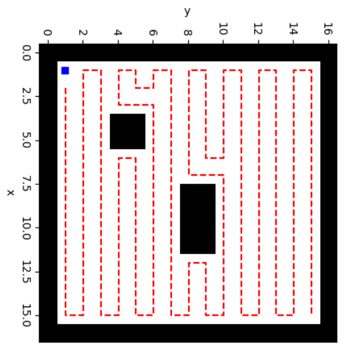

Figure 4.2: Path traced by Algorithm: Two Obstacles

As per our observation, the localization error is around 10cm in the x and y directions. Given that the size of our robot is greater than 10cm, the choice of grid size, as explained in section 2, results in overlapping during the execution of the robot. Thus it guarantees 100% coverage considering the localization uncertainty.

We can keep reducing grid size for more and more coverage. Still, it will keep increasing the inefficiency of this operation, as with decreasing grid size, the portion of the area that is cleaned multiple times will keep increasing.

5 Results

Here is the video of the robot running in the designated environment executing the path obtained with a simple Boustrophedon Algorithm: **click here for video**

Figures 5.1 and 5.2 below show the path generated by the algorithm and the actual path traced by the robot on the field as recorded by storing the instantaneous positions of the robot. Both these figures correspond to the environment without any obstacles. In fig 5.1, the black shaded area represents the obstacles/boundaries of the environment. The red dotted line represents the path traced by the boustrophedon algorithm. In fig 5.2, the black line represents the path of the robot's center, the green

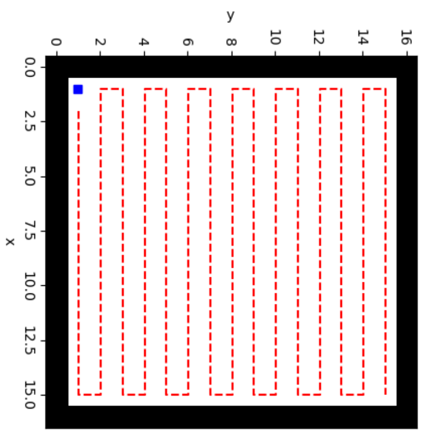

Figure 5.1: Path traced by Algorithm: No Obstacle

trajectory represents the area swept by the vacuum (of size 20cm × 20cm) when the robot is moving along the black line, yellow is the left out free space area, and purple represents the boundary of the map:

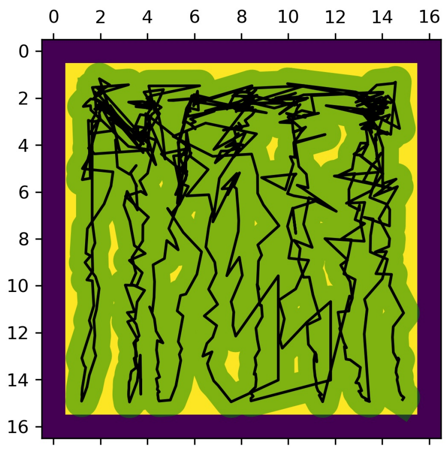

Figure 5.2: Actual path traced by the robot: No Obstacke

The following commands need to be executed in the rosws in order to mimic the results shown in the video.

- source devel/setup.bash
- roslaunch rb5_vision rb_camera_main_ocv.launch
- rosrun april_detection april_detection_node
- rosrun rb5_control hw2_mpi_control_node.py
- python2 fccp_hw5.py
- rosrun rb5_control hw5_sol.py