Maze Solving Bot

Contributors:

Mann Doshi

Dushant Panchbhai

Mithilesh Patil

Harsh Shah

Dhruva

Neha

Omkar

Lukesh

Abhinav

Shantanu

saharsh

Omkar Bhilare

Preeti

sanath

Project Overview :

Micromouse is an autonomous, mobile robot that starts from an initial position in a maze and explores the maze(by themselves) and solves the maze(in this case 16x16). In this project, we have tried to create a simulation to achieve this objective.

In this project, we are making a bot that will first traverse the maze, and then it will suggest the best suitable path to reach the starting point.

For the simulation part, COPPELIASIM and DFS algorithm is used.

- ## Algorithms used:

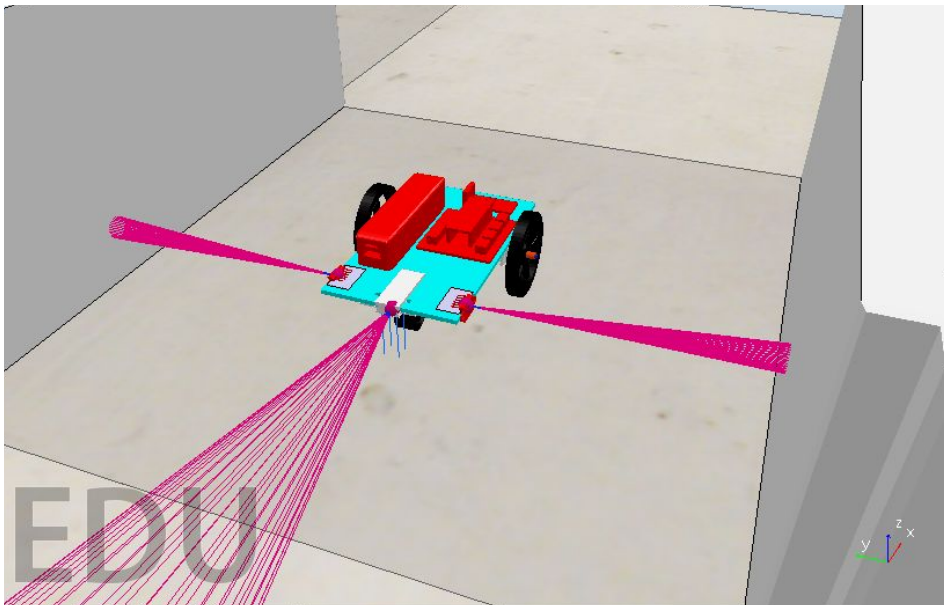
  Some of the algorithms for maze solving we used are :

  1.Depth-First Search (DFS)

  2.wall following algo

- ## Bot design:

- **For DFS**

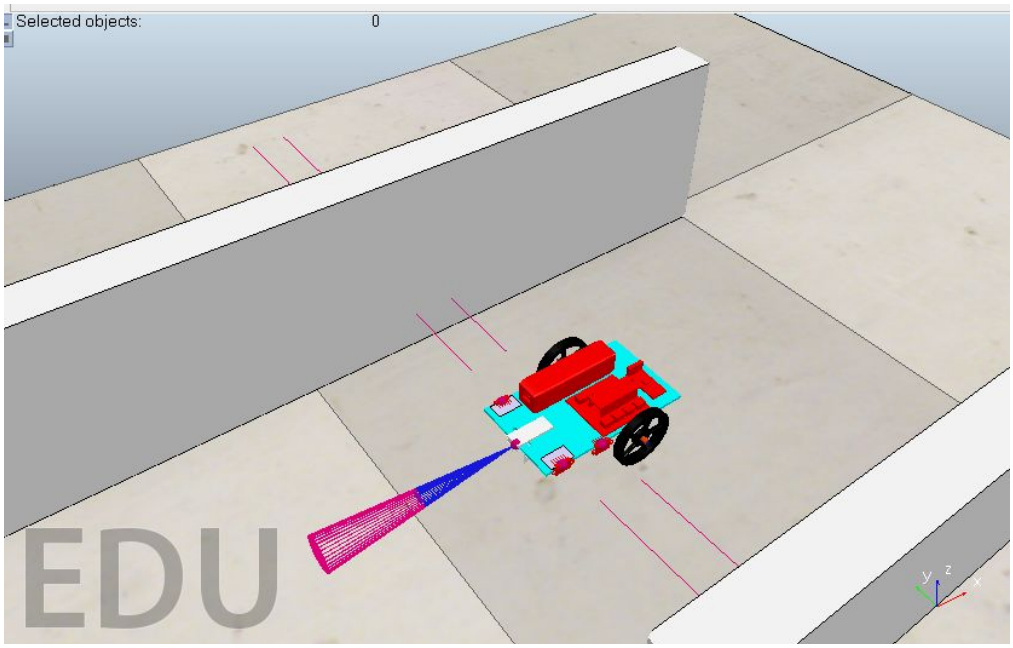


This is the design of the bot which is being used for the dfs algo. It consists of 3 proximity sensors. One at the front and two present at the side of the bot.

Thus this is helpful in determining the presence of an obstacle in the side and front of the bot.

Also one vision sensor is also used which is fixed at the base for determining the end line. Thus when the end line determines, the bot should come to know that it is an endpoint.

- **For wall following**



In this bot, one proximity sensor is used placed at the front and 4 lidar sensors are used which are placed as 2 sensors per side.

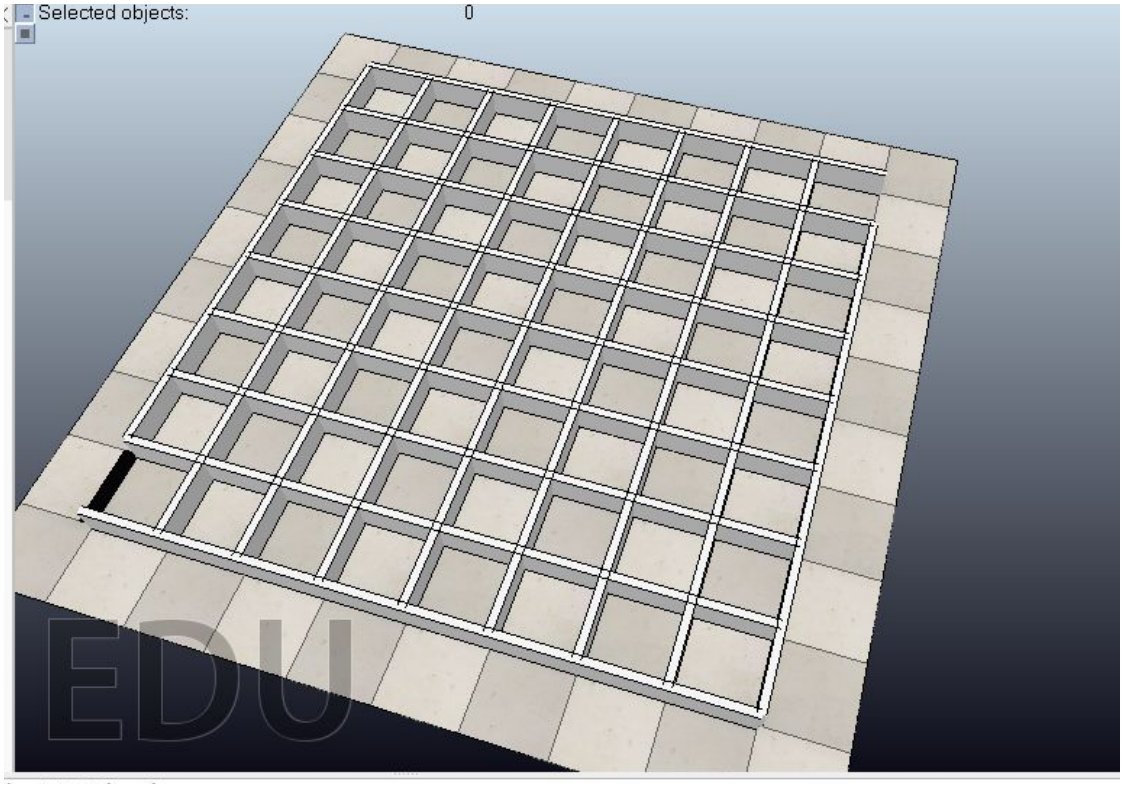
Thus these sensors are used to follow the right wall and also maintain the bot between the wall properly align.

Technology used

- **coppeliaSim**

Creation of maze:

- For the creation of the maze **maze.ttm** file is being used.
- You can get this file in GitHub repository link
- We have to import this file as a model in coppeliaSim.
- This file consists of 1*1 square being aligned in 16*16 square. So, for the creation of the maze, we can delete the required sides of the square and form the maze according to it.



METHOD:

Depth-First Search

In order to figure out how to traverse a maze through code, we first need to understand what depth-first search is The depth-first search (sometimes referred to in this article as DFS) is a graph/tree traversal algorithm that follows a path as far as it can until it either reaches the goal or has nowhere else to go. It's almost like running as far as you can in one direction until you hit a wall.

Before we dive into the algorithm itself there are a few things we need to understand about depth-first search first. There are many approaches and styles to implementing a depth-first search algorithm and a lot of those implementation choices will entirely depend on the problem you're trying to solve. So first let's define some assumptions. These assumptions may seem real straight forward but they're really important in helping us come up with a solution to solve our maze.

Assumptions:

Each maze has a starting point and an ending point.

This assumption is mostly for simplicity's sake. We could handle edge cases all day. Like how there may not be any path that leads to an exit in the maze but that isn't the point of this project. Instead, we're going to keep our constraints as close to an actual maze as possible. Surely an actual maze would have an exit. :)

Mazes are composed of walls.

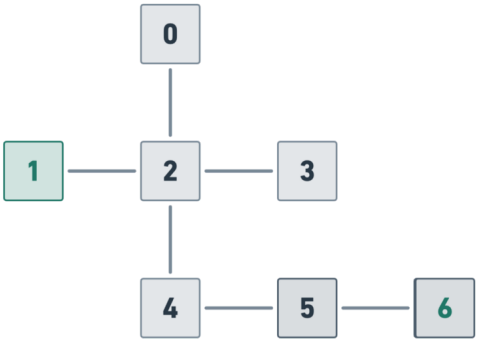
These walls will be considered untraversable. This means we, unfortunately, can't code our solution in a way that bypasses walls. So unless we've recently found a way to walk through matter, we've defined a constraint. Therefore, we need to find some way to alert the computer where walls exist or rather what paths are available at any given point in the maze. Here the availability of the paths is determined by proximity sensors on each side of the bot and in the front.

We only care about finding the first available path through the maze.

This third and final assumption we're making may seem a bit shocking but it's important to understand that in our implementation of DFS we aren't concerned with finding the shortest path. We only care that there is a path and that we're able to return in.

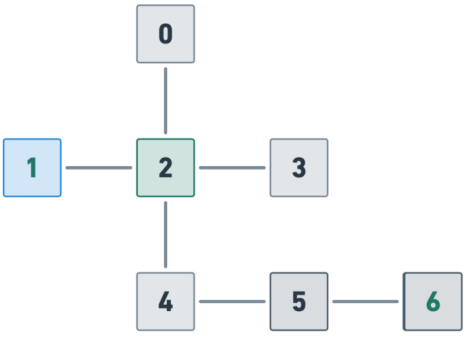
Traversing

Graphs typically don't have a designated starting point but in order to perform DFS, we have to start somewhere so we'll start with Node 1.



Step 1

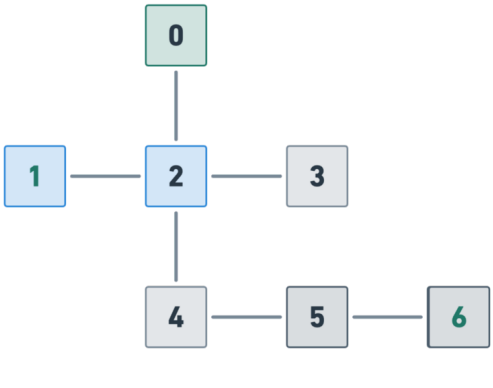
What's happening is that we call DFS on Node 1 which puts it on the call stack. We mark Node 1 as visited so that we don't accidentally evaluate the node again as that could lead to us never finding a path! We then check to see if Node 1 is equal to the node we're looking for (Node 6.) It isn't so we move on to Node 1's adjacency list. Every node holds a list of references to all nodes that they are connected to. So we see that Node 2 is connected to Node 1 and move on to the next step!

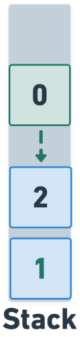


Step 2

The steps are similar. We mark the node as visited, then check to see if it's equal to the node we're looking for. It's not so we call DFS on Node 2 to go even

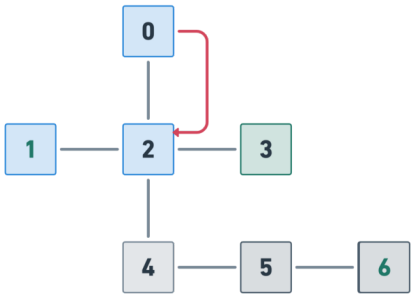
deeper into the graph.Now we look at Node 2's adjacency list. Visually, we can see that Node 2 is connected to node 1, 0, 3 and 4. That's four edges meaning four different paths we can take. So how do we choose? Well, in all honesty, the actual node we choose doesn't matter. The order is entirely dependent on how we stored our nodes in the adjacency list.

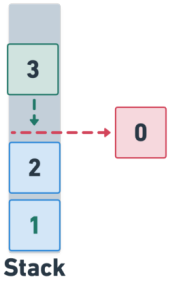




Step 3

Now that we've moved onto Node 0 we go through the same routine as before. We mark it as visited, check if it's equal to Node 6 and then call Depth-First Search on Node 0 to try and move deeper into the maze. However, you may have noticed that the only connector Node 0 has is to Node 2 which we've already visited. So what happens now? Well, when we call DFS on Node 0 we've hit a point where we can't go further down a path so no more function calls (nodes) are added to the stack. Instead, we have to do something called **backtracking**.

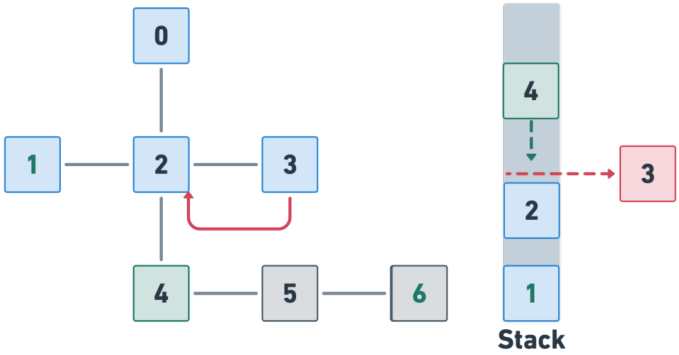




Step 4

Node 1 and *Node 2* are still on the stack because they are still part of a potential path. Once we hit Node 0 though no other paths exist. This means we're done with the DFS call on *Node 0* and hence we have to backtrack to *Node 2*

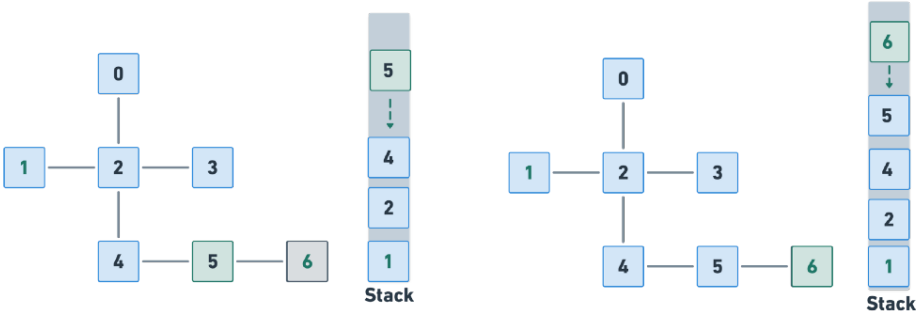
in order to progress deeper in the structure. So we look at the next node in 2's adjacency list and begin to evaluate *Node 3.*



Step 5

Now we do more of the same. We end the DFS call on *Node 3* and thus remove it from the stack. We hop back to 2's adjacency list and begin to evaluate *Node 4*. We notice that *Node 4* is connected to another node we haven't traversed yet and begin the DFS cycle again!

Step 6 and Step 7



Much like before for these figures we visit, compare and call DFS on each node. Only once we hit Node 6 does our goal check finally become true! Now if you look at the call stack in figure 9 you'll notice that it contains a path from Node 6 all the way to Node 1(6 → 5 → 4 → 2 → 1.)All that thanks to backtracking! Now that the function has finally evaluated to true we can begin to remove all our DFS calls from the stack and print off the values. Thus giving us our path!

Wall following algorithm:

The best-known rule for traversing mazes is the *wall follower*, also known as either the *left-hand rule* or the *right-hand rule*. If the maze is *simply connected*, that is, all its walls are connected together or to the maze's outer boundary, then by keeping one hand in contact with one wall of the maze the solver is guaranteed not to get lost and will reach a different exit if there is one; otherwise, the algorithm will return to the entrance having traversed every corridor next to that connected section of walls at least once.

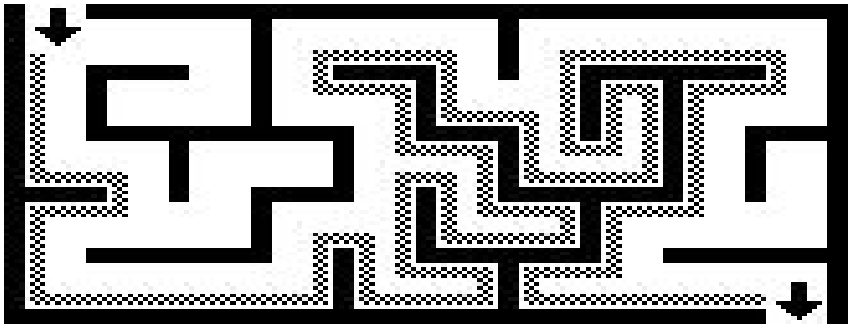
Another perspective into why wall following works is topological. If the walls are connected, then they may be deformed into a loop or circle. Then the wall following reduces to walking around a circle from start to finish. To further this idea, notice that by grouping together connected components of the maze walls, the boundaries between these are precisely the solutions, even if there is more than one solution (see figures on the right).

If the maze is not simply-connected (i.e. if the start or endpoints are in the center of the structure surrounded by passage loops, or the pathways cross over and under each other and such parts of the solution path are surrounded by passage loops), this method will not reach the goal.

Another concern is that care should be taken to begin wall-following at the entrance to the maze. If the maze is not simply-connected and one begins wall-following at an arbitrary point inside the maze, one could find themselves trapped along a separate wall that loops around on itself and containing no entrances or exits. Should it be the case that wall-following begins late, attempt to mark the position in which wall-following began Because wall-following will always lead you back to where you started, if you come across your starting point a second time, you can conclude the maze is not simply-connected, and you should switch to an alternative wall not yet followed. See the *Pledge Algorithm*, below, for an alternative methodology.

Wall-following can be done in 3D or higher-dimensional mazes if its higher-dimensional passages can be projected onto the 2D plane in a deterministic manner. For example, if in a 3D maze "up" passages can be assumed to lead northwest, and "down" passages can be assumed to lead southeast, then standard

wall following rules can apply. However, unlike in 2D, this requires that the current orientation is known, to determine which direction is the first on the left or right.



- **comparison:**

dfs	Wall following
It is fast	It is slow as compared to dfs
applicable in a maze containing islands	Not applicable in a maze containing islands
Cannot guarantee to find a solution.	If maze has no island then a solution would be found for sure.

EXPERIMENT AND RESULTS:

- **Procedure followed:**

Both the type of algorithm i.e DFS and Wall following algorithm is being followed over here.

- **Trial and errors performed:**

Various trials and errors are being performed for the stabilization of the bot.
In the dfs algorithm, the bot sometimes used to take a very long turn and sometimes short. This had led to the error in the traversing of the maze and thus the bot got stuck in the middle of the path. Also in the wall following algorithm, we faced errors for keeping the bot between the walls of the maze.

- **Results:**

Youtube Link for the DFS scene 1:

https://youtu.be/5XY8LEHnnqk

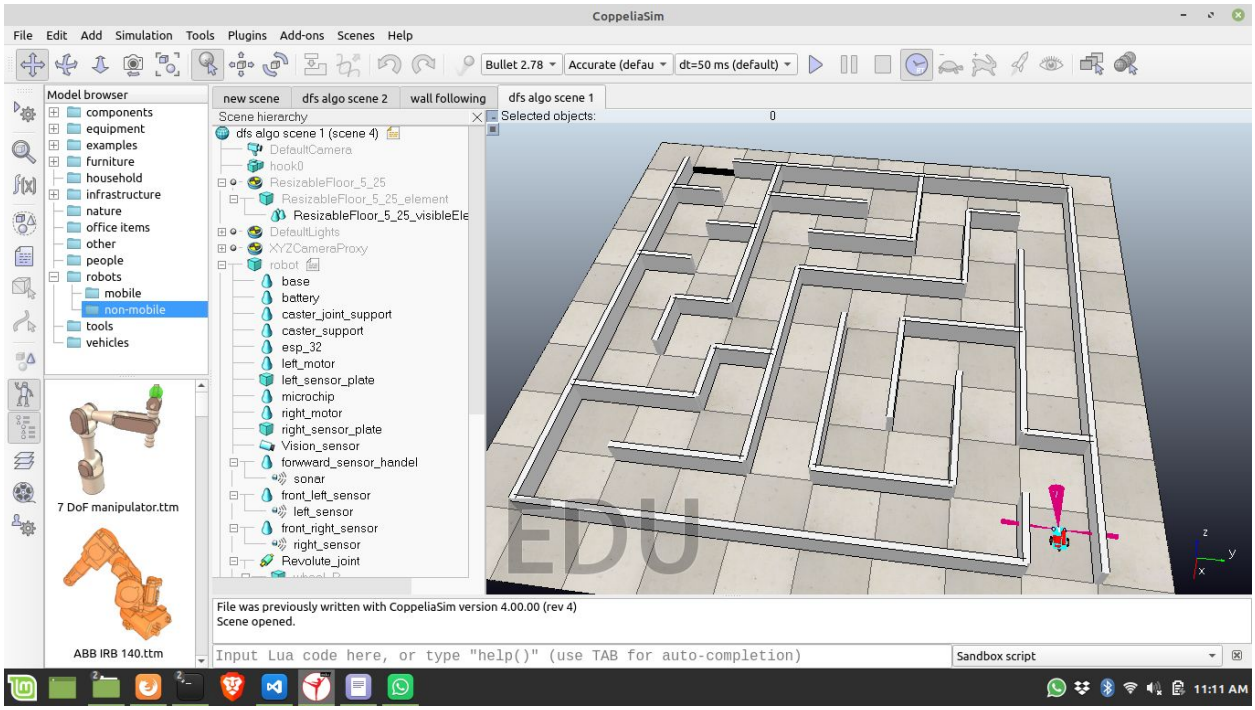
Youtube Link for the DFS scene 2:

https://youtu.be/F3ynhaH-cjE

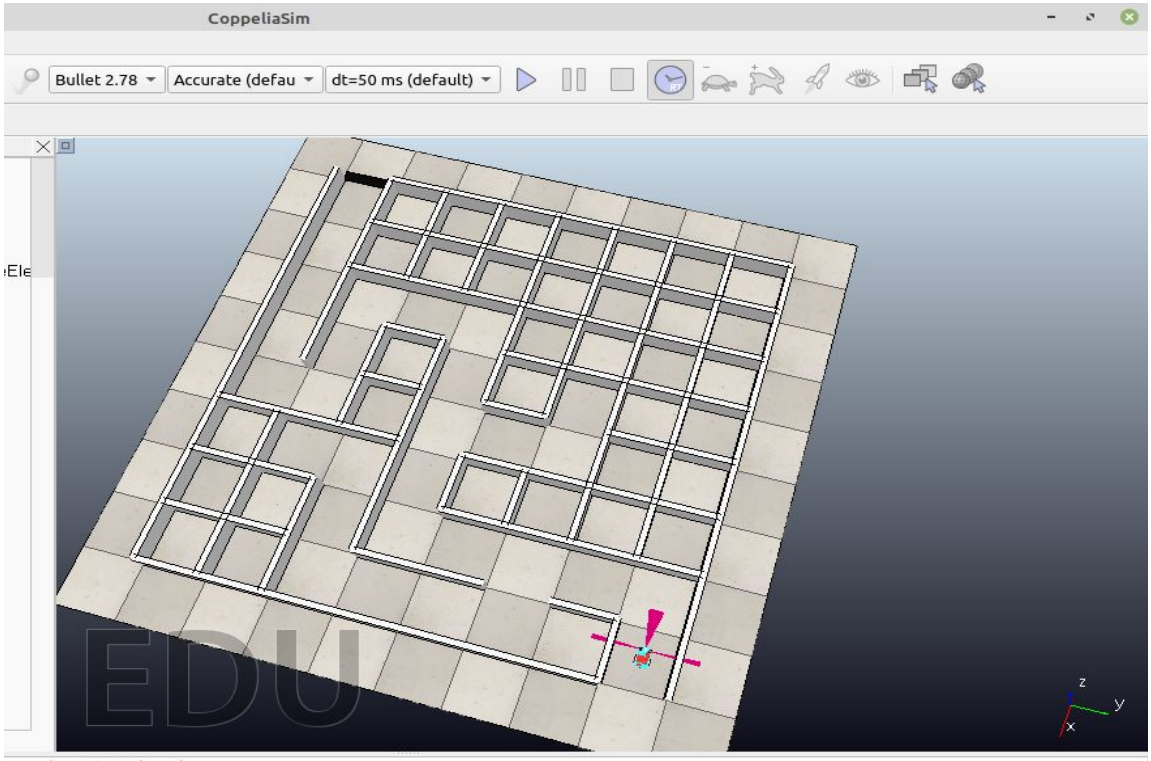
Youtube Link for the wall following algorithm:

https://youtu.be/yTe2L4O4kTQ

Scene 1


CoppeliaSim
File Edit Add Simulation Tools Plugins Add-ons Scenes Help
Bullet 2.78
Accurate (defau
dt=50 ms (default)
Model browser
components
equipment
examples
furniture
household
infrastructure
nature
office items
other
people
robots
mobile
non-mobile
tools
vehicles
7 DoF manipulator.ttm
ABB IRB 140.ttm
new scene
dfs algo scene 2
wall following
dfs algo scene 1
Scene hierarchy
dfs algo scene 1 (scene 4)
DefaultCamera
hook0
ResizableFloor_5_25
ResizableFloor_5_25_element
ResizableFloor_5_25_visibleEle
DefaultLights
XYZCameraProxy
robot
base
battery
caster_joint_support
caster_support
esp_32
left_motor
left_sensor_plate
microchip
right_motor
right_sensor_plate
Vision_sensor
forward_sensor_handel
sonar
front_left_sensor
left_sensor
front_right_sensor
right_sensor
Revolute_joint
Selected objects:
0
EDU
File was previously written with CoppeliaSim version 4.00.00 (rev 4)
Scene opened.
Input Lua code here, or type "help()" (use TAB for auto-completion)
Sandbox script
11:11 AM

Scene 2


CoppeliaSim
Bullet 2.78
Accurate (defau
dt=50 ms (default)
EDU

CONCLUSION AND FUTURE WORK :

- Efficiency and accuracy we are getting in the results

  For the scenes we have shown we are getting good accuracy as compared to the other scenes. But still, there are some instances at these scenes when the bot is getting off track. So there are some instances where the bot is in the way of colliding with the wall.

- What did we achieve:

  Until now we have completed the simulation part of the bot in and applied two algorithms in it. There is a lot to achieve further :)

- Future aspects of the project:

  1. learning ESP 32 functioning

  2. making real bot and implementing DFS algorithm on it

  You can refer the below link for todo list:

  https://drive.google.com/file/d/19-YyoJP9NEUblplu_qk7S5FMpm9HO7ml/view?usp=sharing

References:

For dfs algo guide:

https://brilliant.org/wiki/depth-first-search-dfs/

For a video tutorial on dfs algorithm :

https://youtu.be/W9F8fDQj7Ok

For the TODO list:

https://drive.google.com/file/d/19-YyoJP9NEUblplu_qk7S5FMpm9HO7ml/view?usp=sharing

GIThub repository:

https://github.com/dushantpanchbhai/sra.git