ENPM809Y FINAL PROJECT INTRODUCTORY ROBOT PROGRAMMING MAZE SOLVER USING DFS

By Group 4
Karan Sutradhar - UID: 117037272
Sudharsan Balasubramani - UID: 116298636
Sai Praveen Bhamidipati - UID: 117023640
Ashwin Prabhakaran - UID: 117030402

May 13, 2020

1 INTRODUCTION

The basic aim of the project is to implement a Depth-First Search algorithm which allows the robot to navigate through the maze from the start position to goal position. The exploration will follow the following four directions.

1) Down.
2) Right.
3) Up.
4) Left.

For this we have created some C++ class methods which contains the implementation of Depth-First Search algorithm. These class methods are responsible for driving the robot through the maze in all the four directions.

DEPTH-FIRST SEARCH - DFS

1) Depth-First Search (DFS) is an algorithm for traversing or searching tree or graph data structures.
2) The algorithm starts by selecting some arbitrary node as the root node and explores as far as possible along each branch.
3) The implementation of DFS is similar to that of BFS, but differs in two ways,
a) It uses stack instead of queue.
b) It delays checking whether a vertex has been discovered until the vertex is popped from the stack rather than making this check before adding the vertex.
4) After the goal is reached, it shows a path from the start node to the goal node.

So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally print the nodes in the path.

ALGORITHM

1) Create a recursive function that takes the index of node and a visited array.
2) Mark the current node as visited and print the node.
3) Traverse all the adjacent and unmarked nodes and call the recursive function with index of adjacent node.

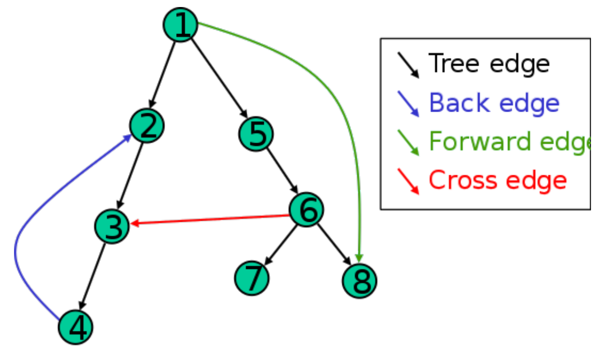


Figure 1: Depth First Search - DFS
https://en.wikipedia.org/wiki/Depth-first_search

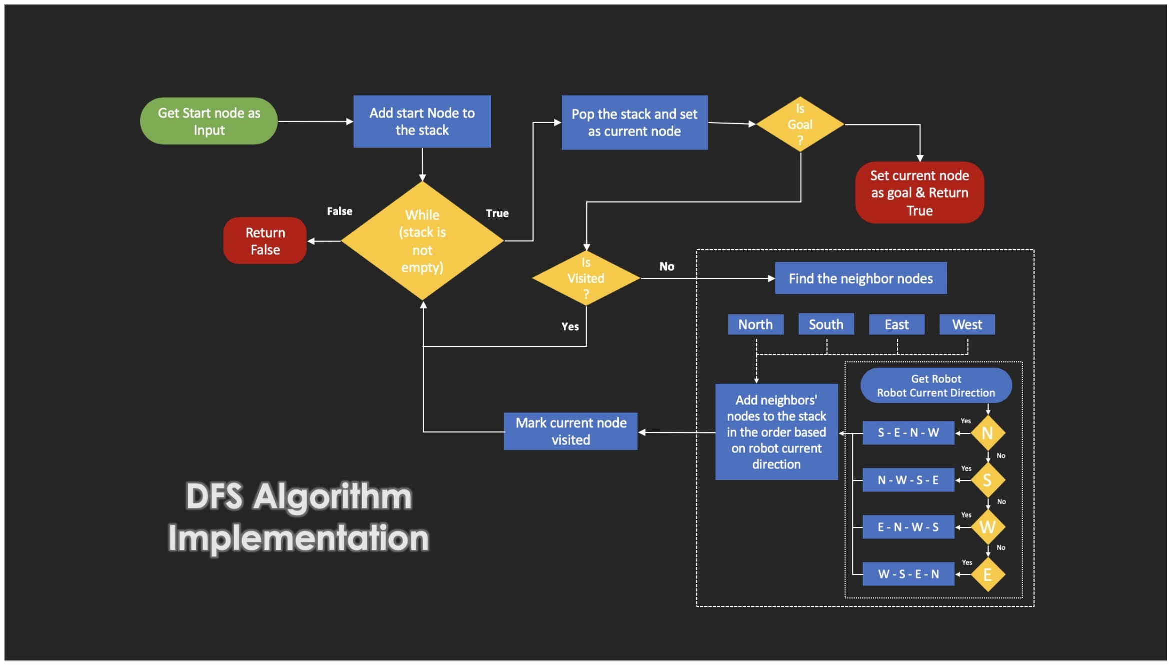

Get Start node as Input
Add start Node to the stack
Pop the stack and set as current node
Is Goal ?
Set current node as goal & Return True
False
True
Return False
While (stack is not empty)
Is Visited ?
No
Yes
Find the neighbor nodes
North
South
East
West
Get Robot Robot Current Direction
Add neighbors' nodes to the stack in the order based on robot current direction
Mark current node visited
S - E - N - W
N - W - S - E
E - N - W - S
W - S - E - N
N
S
W
E
Yes
No
DFS Algorithm Implementation

Figure 2: Pseudo code for Depth First Search Implementation

2 BACKTRACKING

Step 1:

Create two different node arrays, one with a local scope and the other that lives as long as the program runs.
The one with the local scope lives only until the DFSAlgorithm() exits. Everytime the DFSAlgorithm() is called, the node array with the local scope will be reset.
The node array with local scope is used to back track to robot's current position.
The larger scope node array will be used too back track to the initial start node position.

Once a node is already visited, it's parent node doesn't change even if the DFSAlgorithm() is run again.
This way we can effectively do a local search and still back track to the initial start position once the robot reaches the goal node.

Step 2:

Call Backtrack(current node, node array). We need to use the node array with local scope until the robot reaches the goal node.

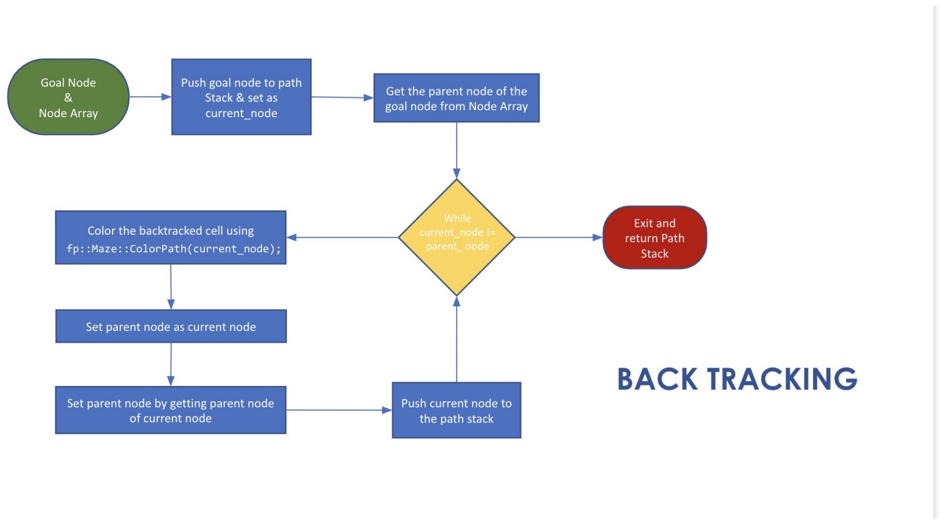
Working:

Once the robot reaches the goal, the current node will be the goal node.

i) Add goal node to the stack.
ii) Recursively find the parent node and add it to the stack and make parent node as current node.'
iii) When the current node's parent equals the current node, we exit the loop and return the path stack.

Navigation:

Using the local path stack, we pop the last two nodes and compute the direction to move and execute it.



Goal Node & Node Array
Push goal node to path Stack & set as current_node
Get the parent node of the goal node from Node Array
While current_node != parent_node
Exit and return Path Stack
Color the backtracked cell using fp::Maze::ColorPath(current_node);
Set parent node as current node
Set parent node by getting parent node of current node
Push current node to the path stack
BACK TRACKING

Figure 3: Backtracking flow chart

PSEUDO CODE

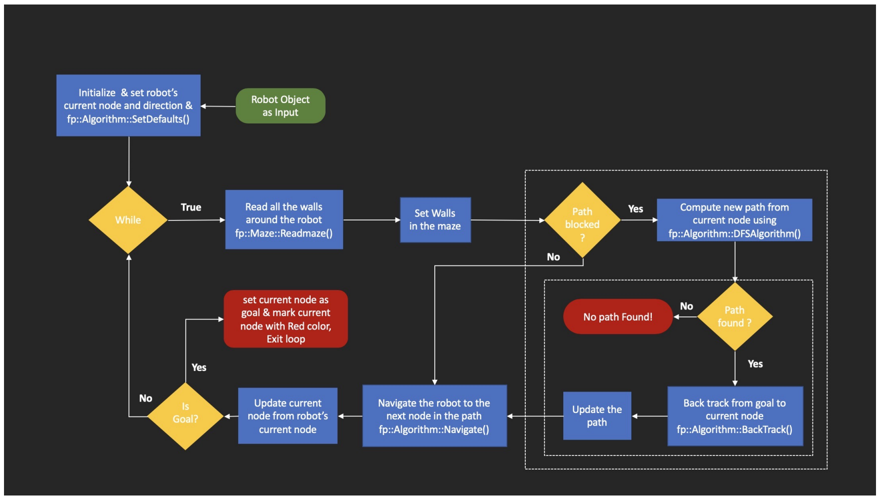

Initialize & set robot's current node and direction & fp::Algorithm::SetDefaults()
Robot Object as Input
While
True
Read all the walls around the robot fp::Maze::Readmaze()
Set Walls in the maze
Path blocked ?
Yes
No
Compute new path from current node using fp::Algorithm::DFSAlgorithm()
No path Found!
No
Path found ?
Yes
set current node as goal & mark current node with Red color, Exit loop
Yes
No
Is Goal?
Update current node from robot's current node
Navigate the robot to the next node in the path fp::Algorithm::Navigate()
Update the path
Back track from goal to current node fp::Algorithm::BackTrack()

Figure 4: Pseudo Code for maze solver implementation

3 CLASS METHODS

The class methods we implemented in the project contains the following files which are required for the execution.

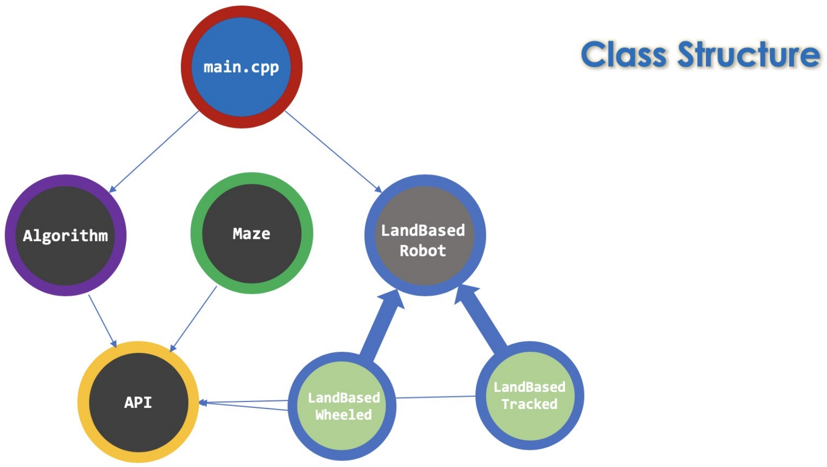


Figure 5: Class methods structure

API

1) The API class is implemented so as to interact with the micromouse controller (MMS). The methods we included in this are wallFront(), wallRight(), wallLeft(). These methods are referred to as boolean functions.
2) These functions will return true value if any wall is detected in their way.
3) The other functions include moveFoward(), moveRight(), moveLeft(). These are the functions used for the directions which help the the robot to navigate throughout the maze.
4) The setWall() is used for setting a wall at a given position.

All the class methods included in the API class are public and can directly call those methods in our program.

LAND BASED ROBOT

1) The robots used in this project are the mobile robotis.
2) The LandBasedTracked and LandBasedWheeled are the mobile robotics arms and are considered as concrete classes which are derived from the base class LandBasedRobot which is the abstract class.
3) These classes follow the dynamic polymorphism as the methods are virtually overriden.

The LandBasedRobot contains the attributes related to speed, width, length, height and capacity of the robot. The methods included are useful for the robot navigation throughout the maze. The methods we inlcuded are,

i) GetDirection() - Gets the direction of the robot in the maze.
ii) MoveForward() - Moves the robot forward.
iii) TurnLeft() - Rotates the robot 90 degrees counter-clockwise.
iv) TurnRight() - Rotates the robot 90 degrees clockwise.

The TurnRight() and TurnLeft methods are only used to rotate the robot. MoveForward() method is used to move the robot again.

MAZE

1) The maze class is responsible for the information contained about the maze which is to be explored.
2) The methods inlcuded in this maze file are ReadMazeFile() and colorpath(). The ReadMazeFile() updates the current position of the maze matrices and the colorpath() sets color to the path which the robot has to follow in the given maze.

The maze we created has the dimensions 16×16. The maze is surrounded by walls on all the sides and also in between the maze which changes the direction of the robot if any wall is present in its way. As the robot has no knowledge about the walls, we used the API methods to check for walls. The robot always faces the north position when the simulation starts. The starting point of the robot position is always (0,0). The goal position is present at the center of the maze and the goal positions may be anywhere between (7,7), (7,8), (8,7), and (8,7).

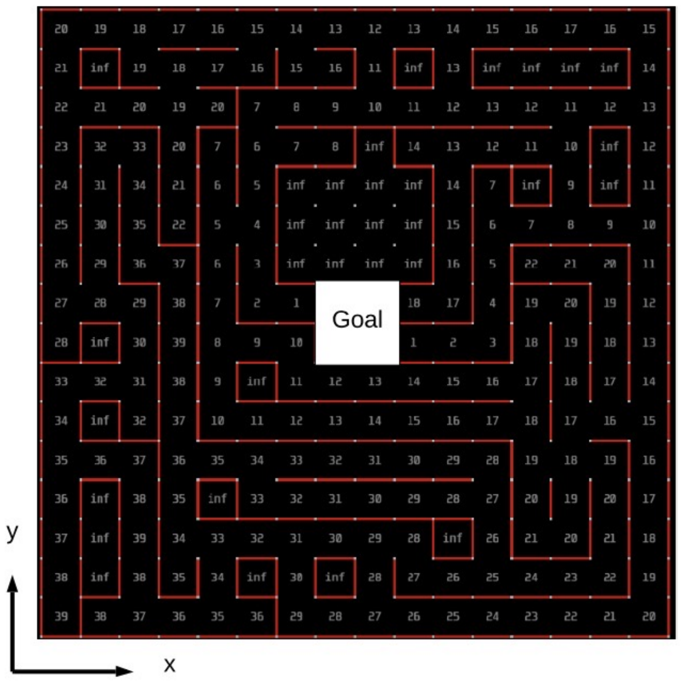
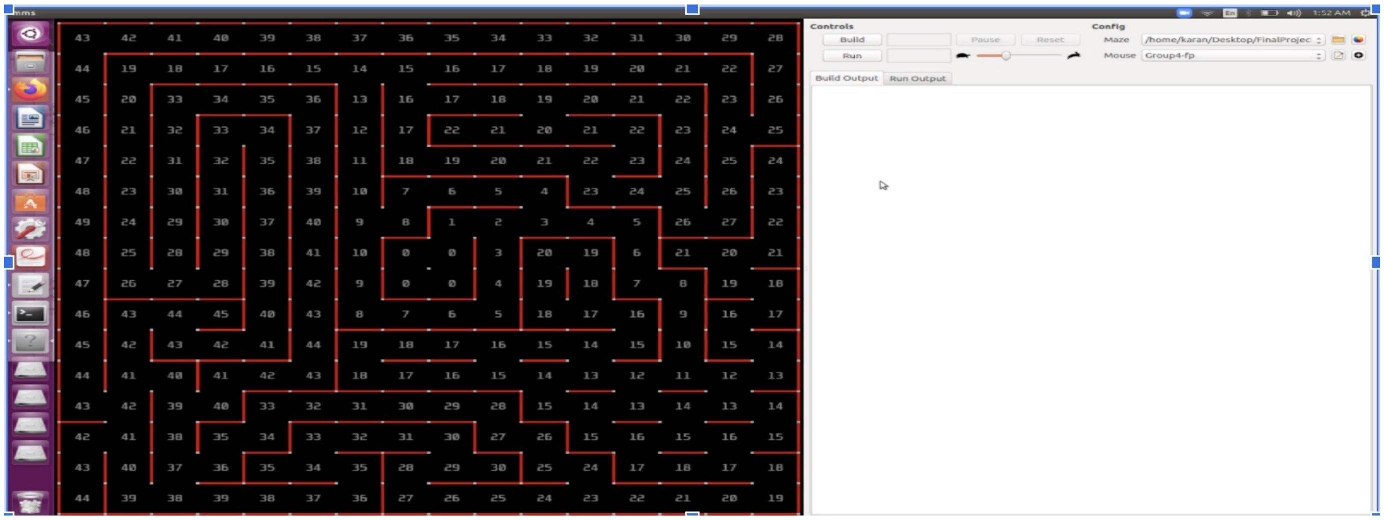


Figure 6: Maze - Coordinate system

4 MICROMOUSE SIMULATOR

In this project we used a micromouse simulator for the simulation. The micromouse simulator is an open source simulator used for the visualization of the algorithm. The API class methods help in the interaction between the program and the simulator as the API class includes information on the width, height, directions of the robot and also position of the walls.

The robot needs to keep track of where it is, discover walls as it explores, map out the maze and detect when it has reached the goal. Having reached the goal, the robot will typically perform additional searches of the maze until it has found an optimal route from the start to the finish. Once the optimal route has been found, the robot will run that route in the shortest possible time.



Maze /home/karan/Desktop/FinalProjec
Mouse Group4-fp

Figure 7: MMS Simulator

5 PROJECT EXECUTION

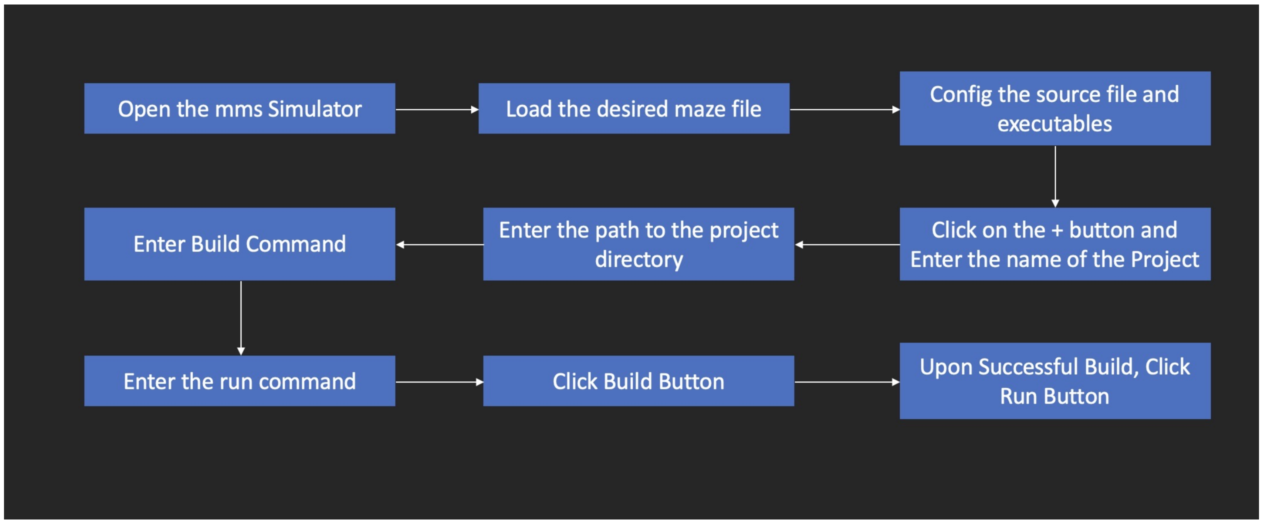

Open the mms Simulator
Load the desired maze file
Config the source file and executables
Click on the + button and Enter the name of the Project
Enter the path to the project directory
Enter Build Command
Enter the run command
Click Build Button
Upon Successful Build, Click Run Button

Figure 8: Project Execution

Steps to run this program:

1) Visit this Github link https://github.com/mackorone/mms#building-from-source to install Micromouse simulator.
2) Build the simulator using the following command: /user cd mms/src /user, qmake and make.
3) Run the simulator using the following command

4) Parameters to be changed in the simulator:
Name:- Input your desired name.
Directory:- ../../Final-Project-Group4/Final˙Project/src Build Command:- g++ -std=c++14 API/api.cpp
LandBasedRobot/landbasedrobot.cpp LandBasedTracked/landbasedtracked.cpp LandBasedWheeled/landbasedwh
Maze/maze.cpp Algorithm/algorithm.cpp main.cpp
Run Command:- ./a.out.
5) Run the Simulator.

6 CONTRIBUTIONS

1) Karan Sutradhar - Blueprint of Project/ Work Allotment and Coordinating-Setting up zoom meetings-Project timeline/Coding-DFS and Class Structure/ Presentation.

2) Sudharsan - POC with Professor for Doubts/Coding-DFS and Class Structure/ Compiling Codes/ Error Handling and Testing/ Flow Charts for Presentation.

3) Sai Praveen Bhamidipati - Background data collection/ Doxygen file/ Coding- Maze and API classes/ Documentation- comments on codes/ Report and Presentation/Error Handling/ Testing.

4) Ashwin Prabhakaran - Github Repository Maintenance/ Verifying Google Naming Conventions/ Coding- Maze and API classes/ Documentation- comments on codes/ Report/ Presentation/Testing.

7 FUTURE IMPROVEMENTS

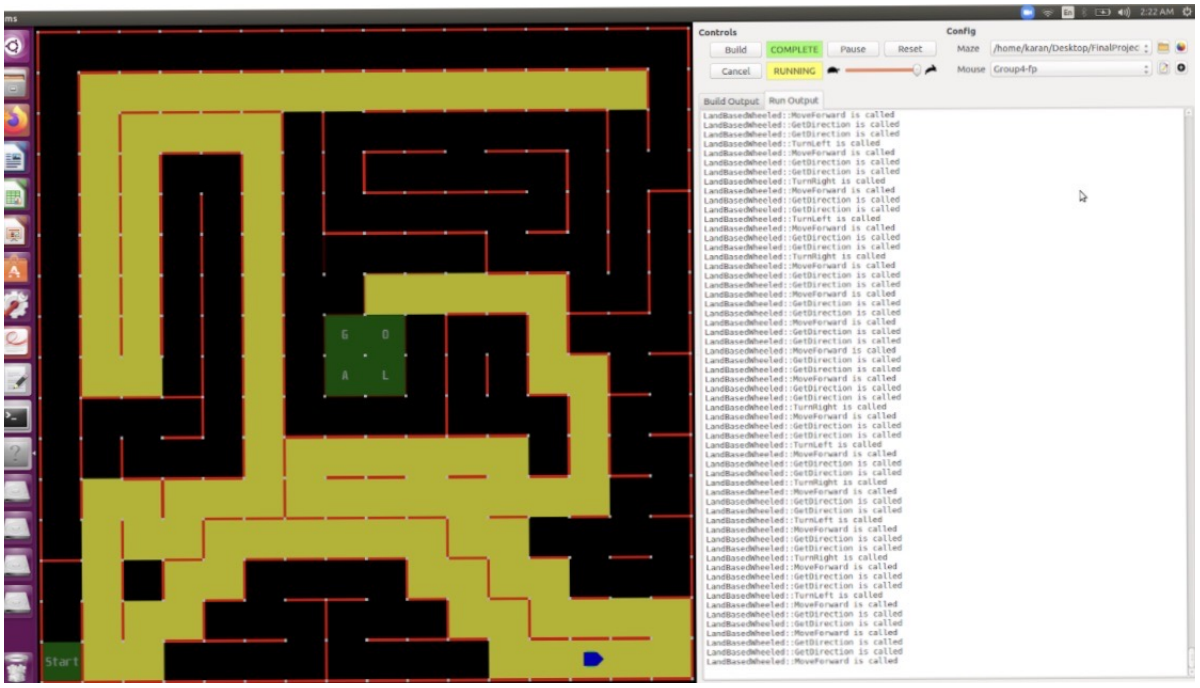
1) We would like to implement other path planning algorithms like BFS, Dijkstra, A* etc for the same maze.
2) For different and difficult mazes we would like to implement a code which runs faster and give optimal results.
3) Changing the Micro-mouse Simulator to add dynamic obstacles on already explored locations.

8 CHALLENGES FACED

1) We had a problem with the backtracking of the code. The robot was failing to backtrack the path. We later implemented it correctly.
2) Usage of raw pointer was an issue. So we used shared pointer instead.

9 RESULTS

1) Successfully implemented DFS Algorithm to solve the maze.
2) OOP based code development.



Maze /home/karan/Desktop/FinalProjec
Mouse Group4-fp

```
LandBasedWheeled::MoveForward is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::TurnLeft is called
LandBasedWheeled::MoveForward is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::TurnRight is called
LandBasedWheeled::MoveForward is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::TurnLeft is called
LandBasedWheeled::MoveForward is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::TurnRight is called
LandBasedWheeled::MoveForward is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::MoveForward is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::MoveForward is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::MoveForward is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::MoveForward is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::TurnRight is called
LandBasedWheeled::MoveForward is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::TurnLeft is called
LandBasedWheeled::MoveForward is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::TurnRight is called
LandBasedWheeled::MoveForward is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::TurnLeft is called
LandBasedWheeled::MoveForward is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::TurnRight is called
LandBasedWheeled::MoveForward is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::TurnLeft is called
LandBasedWheeled::MoveForward is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::MoveForward is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::GetDirection is called
LandBasedWheeled::MoveForward is called
```

Figure 9: Final Result

10 REFERENCES

1) DFS Algorithm : https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm
2) BFS Algorithm: https://en.wikipedia.org/wiki/Breadth-first_search
3) Mms simulator : https://github.com/mackorone/mms
4) Object Oriented Programming : https://beginnersbook.com/2017/08/cpp-oops-concepts/