# Final Project Report

ENPM 809Y – Group 10

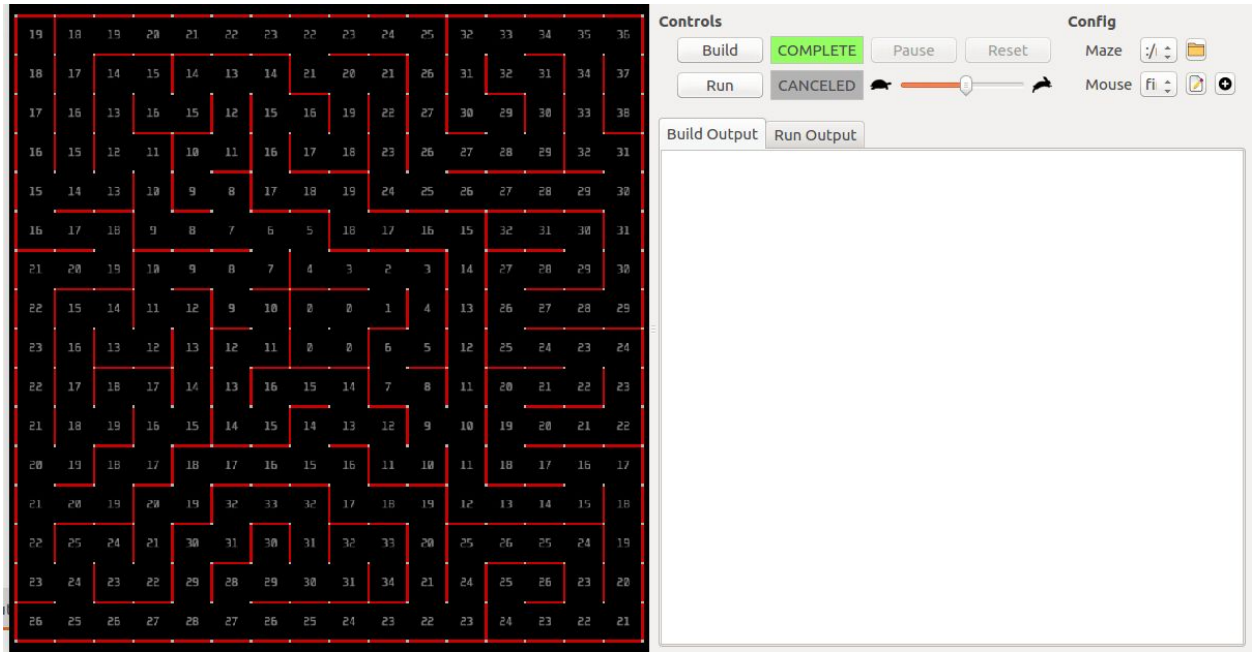


| | |
|---|---|
| Rachith Prakash | 116141468 |
| Prasanna Balasubramanian | 116197700 |
| Alexandre Filie | 110509097 |
| Govind Ajith Kumar | 116699488 |
| Dinesh Kadirimangalam | 116353564 |
| Abhiram Dapke | 116237024 |

# Introduction to the Project

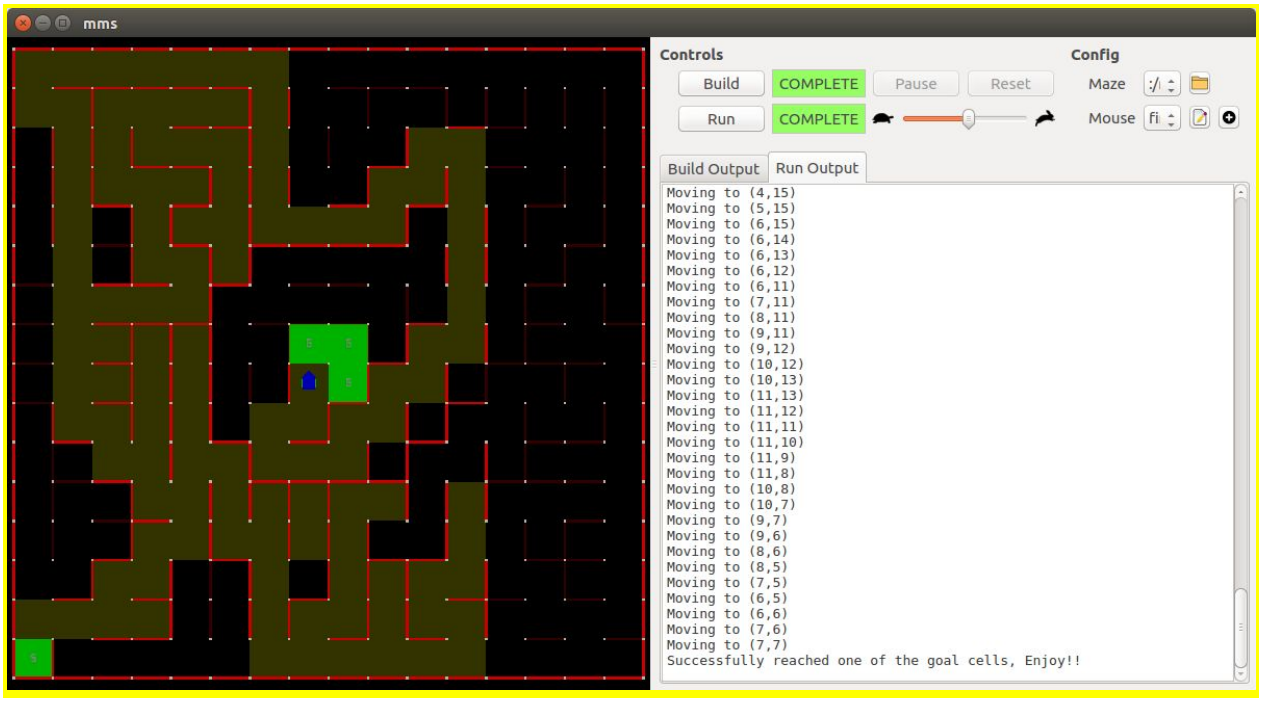
The project is a C++ coding exercise where a micro mouse (hereafter referred to as MMS) simulator environment is used to test the algorithm developed throughout the project. In a nutshell, the mouse/robot starts from a fixed starting location and orientation and has to reach the goals i.e. centers of the map/maze. The map has walls which obstructs the movement of the mouse. The mouse initially does not know the map i.e. locations of the walls, hence it needs to explore and keep a track of the walls it encountered as well as find a new path to goal as and when it encounters a wall(it's current path would be no longer valid). A path may or may not exist.

As you can see below, the start and goal nodes are colored in light/radiant green. The robot creates a path in dark green based on Breadth-First-Search algorithm. As it follows it, it encounters walls, and updated the map with walls(red lines) and generates new path accordingly and tried to follow it. The path visited by the robot can be seen in almost blackish/yellow cells.



# Introduction to the Simulator

The MicroMouseSimulator is a pre-built simulator that can be cloned and used from GitHub. The Simulator interface looks like this :



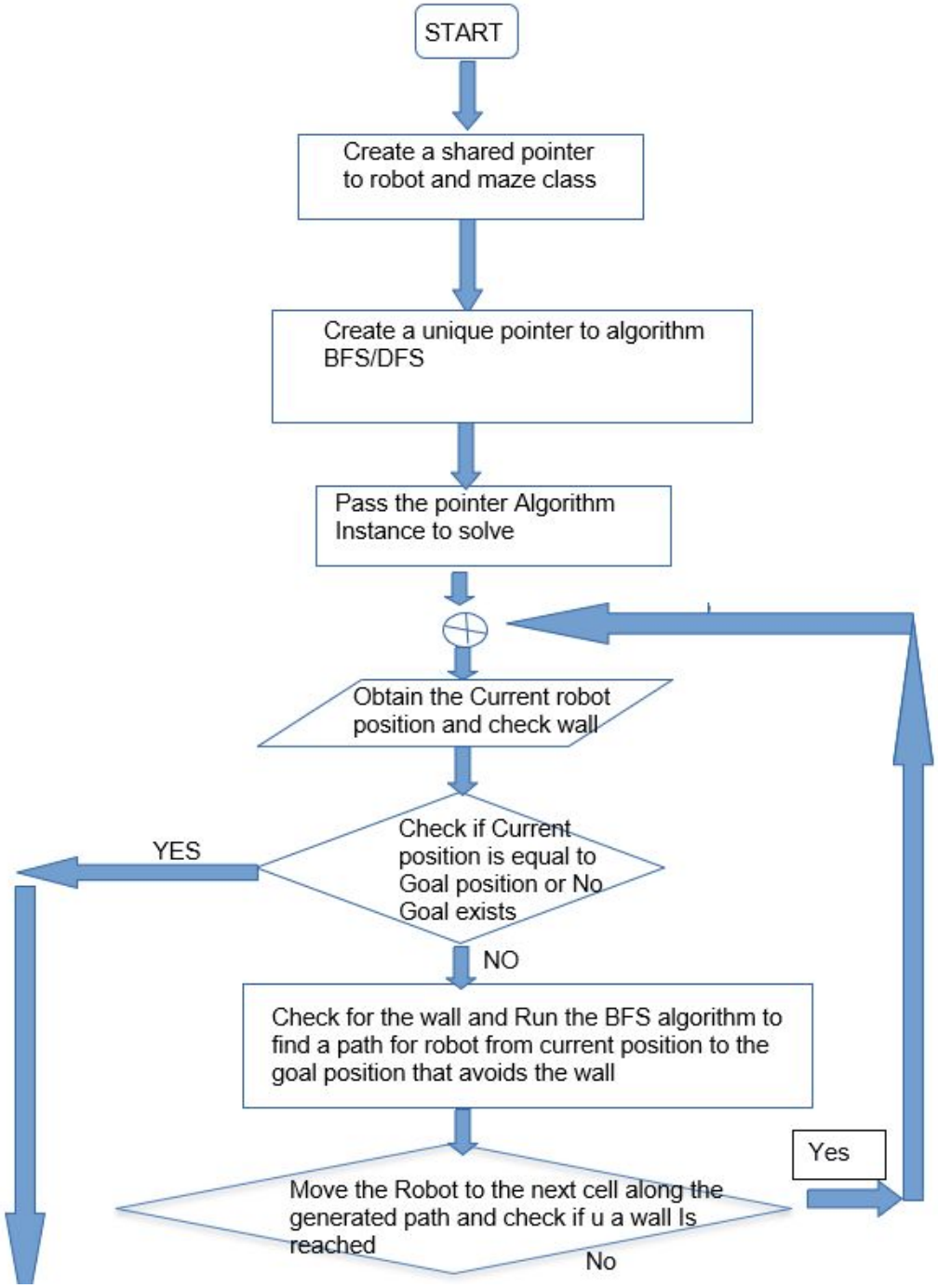

The simulator takes in mazes of size 16 by 16 filled with walls and provides an interface for the mouse on various types of functionalities as described in the API class section. The speed in which the maze can be solved can also be altered using a sliding bar, which will help visualize the whole solution while the mouse finds its way from the beginning to the goal. The mouse always starts from the bottom-left corner of the maze and heading north.

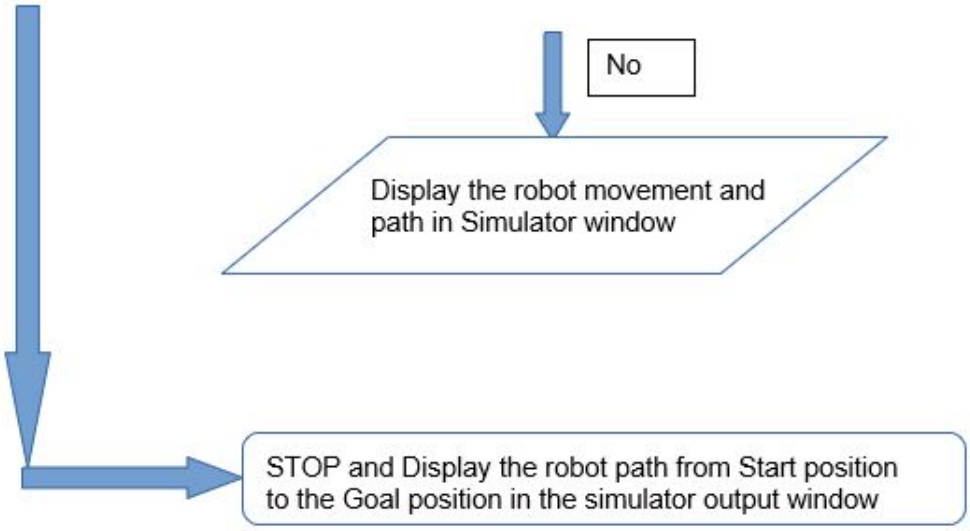
The simulator can be found here : https://github.com/mackorone/mms

# Overview of the Approach

We solve this problem using Dynamic Programming with both Breadth-First-Search and Depth-First-Search algorithm. Various classes are developed that can help bring together the code and the functions. The class titled API is crucial over here to interface with the simulator. The classes, their methods, and their workings are described in great detail below.

FLOW CHART


START
Create a shared pointer to robot and maze class
Create a unique pointer to algorithm BFS/DFS
Pass the pointer Algorithm Instance to solve
Obtain the Current robot position and check wall
Check if Current position is equal to Goal position or No Goal exists
YES
NO
Check for the wall and Run the BFS algorithm to find a path for robot from current position to the goal position that avoids the wall
Move the Robot to the next cell along the generated path and check if u a wall Is reached
Yes
No




# Algorithm

## Breadth-First Search

Breadth-first search (hereafter referred to as BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. It uses a Queue i.e. First In First Out(FIFO) approach.

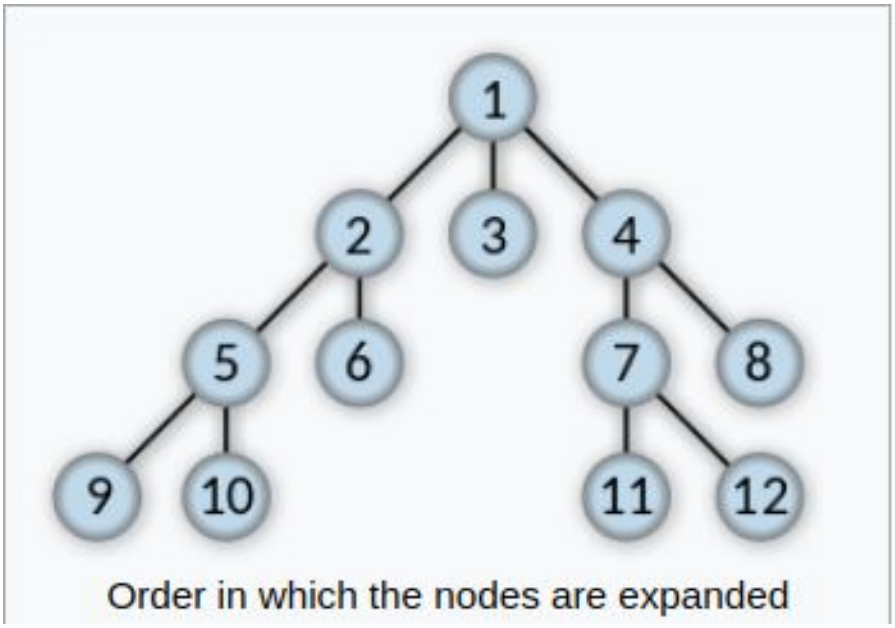



*Figure*: Breadth-First Search spanning tree

Credits:https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

## Depth-First Search

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. It uses a stack i.e. Last In First Out approach.

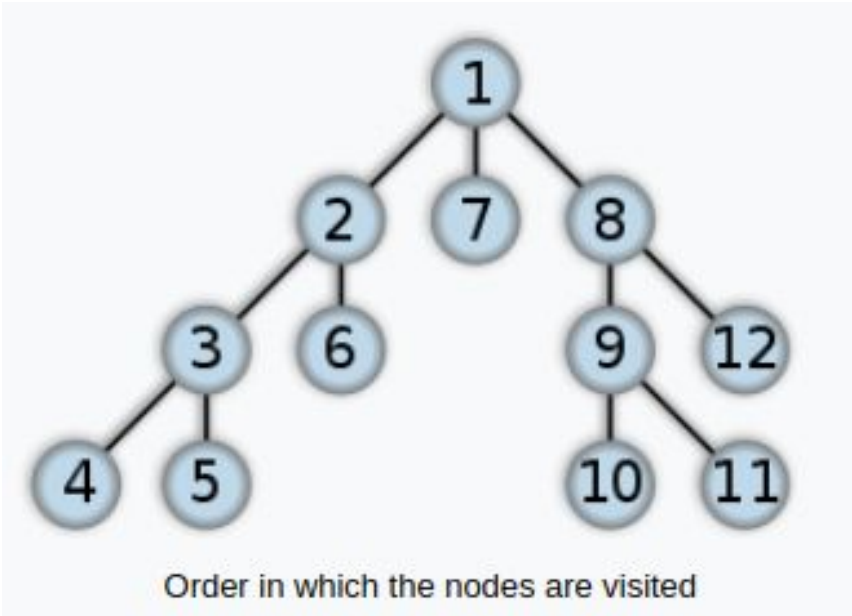


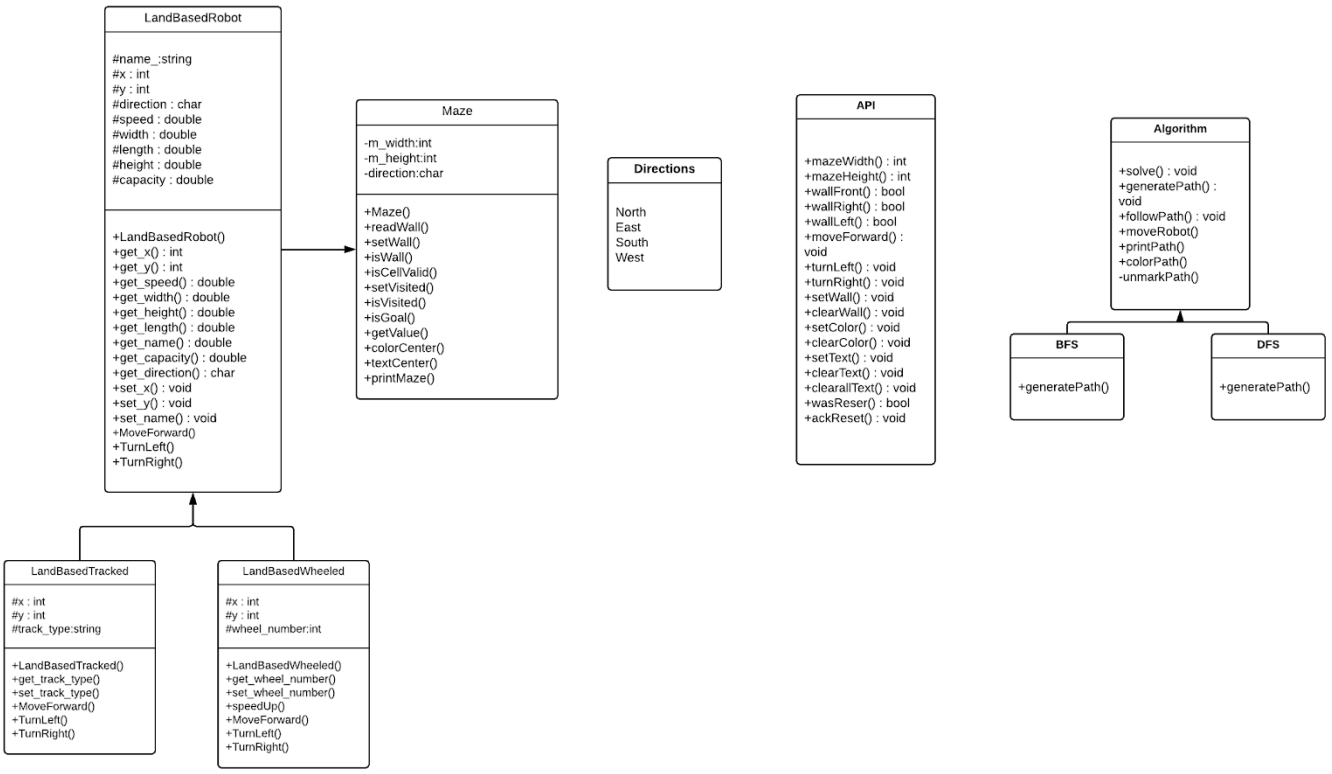
Figure: Depth-First Search spanning tree

Credits: https://en.wikipedia.org/wiki/Depth-first_search

## *Breadth-First Search vs Depth-First-Search* Algorithm?

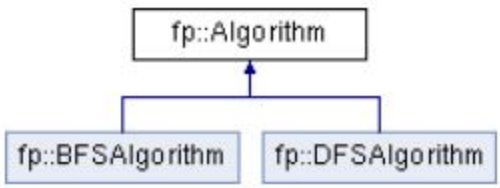
Suppose we are dealing with a manageable limited branching factor and a manageable limited depth. If the searched node is shallow i.e. reachable after some edges from the original source, then it is better to use BFS. On the other hand, if the searched node is deep i.e. reachable after a lot of edges from the original source, then it is better to use DFS. In this scenario, the maze is not too deep but rather shallow, hence if we explore nodes in parallel, goal can be reached earlier as compared to DFS. We have implemented both the algorithms. Based on empirical analysis on 5 mazes provided with the simulator, BFS performed better (in the sense, lesser exploration of the robot) and quicker .

# UML Diagram




# Classes

## Algorithm Class



This is an abstract class, given a robot start position in (x, y) coordinates, it finds a path to the goal position from the start. It has the functions to move the robot along the generated path and color the path. Algorithm calls maze pointers to determine cell locations and wall positions. Wall positions are read at the robot's current position and clear paths determined.

| Method | Function |
|---|---|
| ***Algorithm()*** | Constructor; sets the color and text of initial start location, goal locations. |
| ***followPath*** | This method executes the path generated one by one along with checking the walls at every location it moves to. This invokes moveRobot() at every step. |
| ***moveRobot*** | This method moves the robot from (x,y) to (x1, y1) by interfacing with the mms simulator. |
| ***printPath*** | Prints the generated path for debugging purposes. |
| ***colorPath*** | Here, path is printed in ‘green’ to show how the robot is planning to navigate. |
| ***solve*** | This method invokes rest of the methods and constantly checks if we have reached goal node. |
| ***generatePath*** | Virtual function which derived classes can use to customize their algorithm. |

## BFSAlgorithm Class

As shown above, this class is a derived class of class Algorithm. The inheritance is public inheritance. The only method that needs to be overridden is the generatePath method as described below.

| Method | Function |
|---|---|
| ***generatePath*** | Computes/calculates the path based on BFS technique. Uses queue data structure to |

| | |
|---|---|
| | execute the algorithm. |

## DFSAlgorithm Class

As shown above, this class is a derived class of class Algorithm. The inheritance is public inheritance. The only method that needs to be overridden is the generatePath method as described below.

| Method | Function |
|---|---|
| ***generatePath*** | Computes/calculates the path based on DFS technique. Uses stack data structure to execute the algorithm. |

## API Class

In this class, given a robot start position in (x, y) coordinates and it finds a path to the goal position from the start. It has the functions to move the robot along the generated path and color the path. The functions present in the API class help interface the entire code with the Simulator and its inbuilt functions. The functions present in the API Class and their functions are described below:

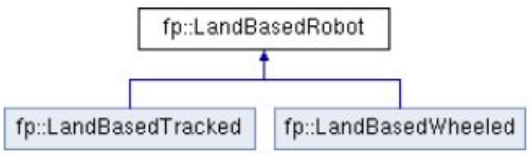
| Method | Function |
|---|---|
| ***mazeWidth*** | Maze width reported from maze file valuation |
| ***mazeHeight*** | Maze height reported from maze file evaluation |
| ***wallFront*** | Boolean verification if a wall is present in front of the current location and heading |
| ***wallRight*** | Boolean verification if a wall is present on the right of the current location and heading |
| ***wallLeft*** | Boolean verification if a wall is present on the left of the current location and heading |

| | |
|---|---|
| ***moveForward*** | An operation to move the robot forward one position |
| ***turnRight*** | An operation to rotate robot clockwise 90 deg |
| ***turnLeft*** | An operation to rotate robot counter-clockwise 90 deg |
| ***setWall*** | sets wall in specified location and direction |
| ***clearWall*** | clears wall from specified location and direction |
| ***setColor*** | sets color to a wall in specified location and direction |
| ***clearColor*** | clears wall colors around a specified location |
| ***clearAllColor*** | clears all wall colors in a maze |
| ***setText*** | sets the text in a maze cell at a specific location |
| ***clearText*** | clears the text in specified maze cell |
| ***clearAllText*** | clears all text in maze cells |
| ***wasReset*** | resets maze and search operation, returns boolean for completion confirmation |
| ***ackReset*** | resets API response to verify external operations completed |

## Direction Class

The direction class consists of a structure to store the possible directions of the robot. There are four possible directions possible in this scenario namely; North, East, South and West.

## LandBasedRobot Class



This is just an abstract class with the general attribute of the robot such as position i.e., x, y, and direction.

| Method | Function |
|---|---|
| ***get_x*** | Accessor to get the robot X position |
| ***get_y*** | Accessor to get the robot Y position |
| ***get_name*** | Accessor to get the robot name |
| ***get_speed*** | Accessor to get the robot speed |
| ***get_width*** | Accessor to get the robot chassis width |
| ***get_length*** | Accessor to get the robot chassis length |
| ***get_height*** | Accessor to get the robot chassis height |
| ***get_capacity*** | Accessor to get the robot payload capacity |
| ***get_direction*** | Accessor to get the robot direction |
| ***set_x*** | Sets x value to robot attribute for X-maze position |
| ***set_y*** | Sets y value to robot attribute for Y-maze position |
| ***set_direction*** | Sets robots heading direction |
| ***set_name*** | Sets name string to robot attribute |
| ***MoveForward*** | An operation to move the robot forward one position |
| ***TurnLeft*** | An operation to rotate robot counter-clockwise 90 deg |
| ***TurnRight*** | An operation to rotate robot clockwise 90 deg |

## LandBasedTracked

Inherits the LandBasedRobot class and has implementations of the virtual functions present in the base class, such as MoveForward(), TurnRight() and TurnLeft(). These functions used to navigate in the algorithm class.

| Method | Function |
|---|---|
| ***LandBasedTracked*** | Constructor for **LandBasedTracked** robot using **LandBasedRobot** as a derived object |
| ***get_track_type*** | Getter for track type pointer |
| ***set_track_type*** | Setter for track type pointer |
| ***MoveForward*** | An operation to move the robot forward one position |
| ***TurnLeft*** | An operation to rotate robot counter-clockwise 90 deg |
| ***TurnRight*** | An operation to rotate robot clockwise 90 deg |
| ***~LandBasedTracked*** | Destructor for **LandBasedTracked** |

## LandBasedWheeled

Similar to LandBasedTracked class, LandBasedWheeled inherits the LandBasedRobot class and has implementations of the virtual functions present in the base class, such as MoveForward(), TurnRight() and TurnLeft(). These functions used to navigate in the algorithm class.

| Method | Function |
|---|---|
| ***LandBasedWheeled*** | The constructor of a wheeled land-based robot with name, coordinates, number, and type of wheels |
| ***get_wheel_number*** | Accessor for the number of wheels on the robot |

| | |
|---|---|
| ***set_wheel_number*** | Sets wheel number to robot attribute |
| ***SpeedUp*** | Implements the speed increases in robot |
| ***MoveForward*** | An operation to move the robot forward one position |
| ***TurnLeft*** | An operation to rotate robot counter-clockwise 90 deg |
| ***TurnRight*** | An operation to rotate robot clockwise 90 deg |
| ***~LandBasedWheeled*** | **LandBasedWheeled** destructor |

## Maze

The Maze class is used to build the maze structure for us to analyse in the BFS algorithms.

| Method | Function |
|---|---|
| ***Maze*** | The Constructor for **Maze** class |
| ***~Maze*** | The destructor for **Maze** class |
| ***isWall*** | This function used to stored wall data in the maze to output boolean indicating presence/absence of wall in a certain direction |
| ***readWall*** | Reads presence of the wall in maze file from a given position and current robot heading |
| ***setWall*** | This function uses simulator **API** to set wall in required direction in the simulator |
| ***colorCenter*** | This function colors the centers of the map where it is defined to be the Goal area |
| ***textCenter*** | This function writes text on the center cells |
| ***setVisited*** | This function sets the visitation status of the cell at a location (x,y) |

| | |
|---|---|
| ***isVisited*** | This function checks the visitation status of the cell at a location (x,y) |
| ***isGoal*** | This function checks whether the cell at a location (x,y) is one of the goal locations |
| ***isCellValid*** | This function checks if the next coordinate (x,y) is valid and is not out of bounds of the map |
| ***getValue*** | This function returns the value of unsigned char stored in a cell (x,y) |
| ***printMaze*** | This function runs *for* loop based on maze width and height to print the maze array in the terminal for debugging purposes |

# Contributions

***Rachith Prakash***

Developed the code for establishing interface with MMS simulator. Developed BFS and DFS algorithm from scratch. Helped in Doxygen as well. Contributed to a few sections of the report and ppt.

***Prasanna Balasubramanian***

Contributed on development of classes, path finding algorithm, Maze interfacing in simulator, testing with newly developed maze configuration including the no goal configuration, documentation with Doxygen and preparation of final report and presentation.

***Alexandre Filie***

Worked on development of path planning algorithms, maze interpretation, and simulator interfacing. Generated Doxygen comments and report for the project. Presented final project presentation to the class.

***Govind Ajith Kumar***

Worked on developing the .cpp and .h files for LandBasedRobot Class, LandBasedTracked Class, LandBasedWheeledClass. Also worked on the MMS Simulator. Additionally, worked on Report writing, Doxygen Documentation for LandBasedRobot Class, LandBasedTracked Class, LandBasedWheeledClass, Direction Class, API Classes.

***Dinesh Kadirimangalam***

Worked on development of BFS algorithm, maze interpretation, development of classes and report writing and generated UML diagram.

***Abhiram Dapke***

Created the powerpoint presentation and a part of the report. Also, contributed in the working of the BFS algorithm.

# Future Improvements

- We could have implemented more path planning algorithms on the robot like Dijkstra and A star to find the optimal path.
- Modify the MMS simulator to provide dynamic obstacles.

# General Feedback on ENPM809Y

The course was a great introduction to the field of programming as well as the language of C++. The course was laid out properly and worked its way through the basics, while strengthening the concepts and slowly increasing the difficulty.

The groups assigned for the projects were very well balanced because of the mix of students with minimal as well as higher level programming skills. The projects assigned were also fun to work

on, while improving the core knowledge. It also helped in working with teams and taught additional skills such as Doxygen Documentation, GitHub as well as the basics of ROS.

The structure of the quizzes also gave sufficient time to work on the course material while preparing for the quizzes. The teaching assistants for the project was helpful in clearing doubts and any concerns in regards to the course.In the future, maybe we could get more projects and have a couple of extra classes for ROS as well.

# References


1. BFS algorithm - https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
2. DFS algorithm - https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/
3. MMS simulator - https://github.com/mackorone/mms
4. C++ Programming book - https://books.goalkicker.com/CPlusPlusBook/
5. Object Oriented Programming concepts - https://beginnersbook.com/2017/08/cpp-oops-concepts/