UNIVERSITY OF MARYLAND

Final Project Report

ENPM809Y SPRING 2020

Pradeep Gopal

Markose Jacob

Srikumar Muralidharan

Rohit Thakur

Group 2

Introduction

Finding the path to the goal location can take a bit of time when we have to do it manually in a huge maze. This programme gives as a solution to this problem and successfully finds the path and moves the robot to the goal location.

The task of the program is to navigate a robot in any 16 X 16 maze from the starting location of the robot (0,0 bottom left of the maze) to the goal location (7,7 or 7,8 or 8,8 or 8,7 , i.e. the center of the maze) without any further input from the user. The search algorithm used in this programme is Depth First Search (DFS). The start and goal locations are hard coded.

DFS is a search technique which is quite popular. DFS algorithm traverses or searches tree or graph data structure. The algorithm starts at the root node and explores as far as possible along each branch before backtracking.

The programme is designed to run on a micromouse simulator which is a small autonomous robot tasked to navigate through a maze. Micromouse simulators are widely used in programming competitions.

Overview of Approach

1) Resources used in the Project

a) C++ 14 (IDE - CLion)

C++ 14 is a very popular programming language which is widely used by programmers in the field of robotics. We used CLion as the IDE as it includes many useful features like automatically identifying potential errors and also suggesting corrections. The IDE is also user friendly which makes it a delight to use.

b) Micromouse simulator (MMS)

Micromouse simulator is a famous simulator which is used in programming competitions worldwide. It contains many different mazes

and the task is to drive the robot to the goal from the start location. The API file has many useful methods which help us to identify walls as the robot moves, set color to the cells, rotate the robot, etc which helps the programmer to solve the maze.

c) Zoom

Zoom was a very useful tool which helped us to work efficiently during these difficult times of lockdown. The screen sharing option and being able to control the laptops of our teammates helped us to communicate are different solutions to all the problems we faced efficiently.

d) Google docs

We decided to use google docs as this allows all of us to work on the report simultaneously.

e) Github

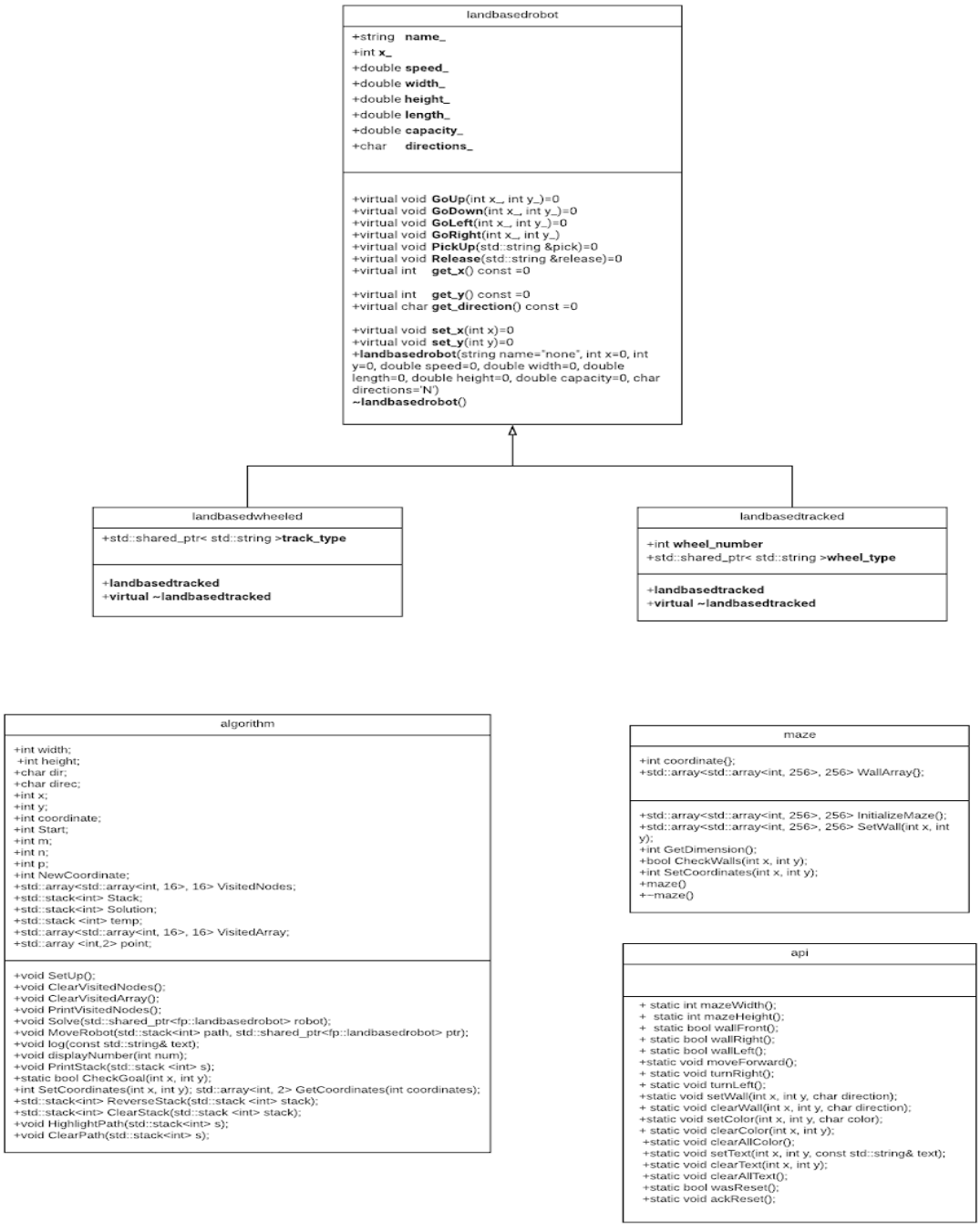
A tool used to make sure everyone is working on the latest code. We had to work in parallel on different branches at the same time so as to be efficient with our limited time for the project and we found Git to be the ideal tool to keep our ideas from not mixing and getting lost. Finally, we extracted the best from each code and stitched them together to generate our final output.

2) Pseudo-code

Algorithm 1: Psuedo-Code for the Final Project

```
Result: Robot of chosen derived class passes through the nodes and
        reaches goal location
initialization - Set walls around the perimeter, color center of maze,
 start point, Optional- choose a single or multiple goal points;
while true do
    clear all tiles other than current and goal node;
    clear stack and visited nodes, load robot coordinates;
    while Path is generated from current to goal do
        if Robot can move down then
            Robot moves Down;
            added to stack;
        else
            if Robot can move right then
                Robot moves Right;
                added to stack;
            else
                if Robot can move up then
                    Robot moves Up;
                    added to stack;
                else
                    if Robot can move left then
                        Robot moves Left;
                        added to stack;
                    else
                        No operation possible for current step;
                        Stack.pop();
                        New Robot Coordinate ← Stack.top();
                        Check visited nodes;
                    end
                end
            end
        end
    end
    if No path from current to goal node then
        Unsolvable case;
        return false;
    end
    Draw path in maze using inbuilt API function;
    Move Robot along defined path using API functions;
    if Robot physically reaches goal then
        Return false;
    end
end
```

3) UML Diagrams For Classes



4) C++ Classes

There are two types of classes in C++, abstract class and concrete class. An abstract class does not have to actually contain the implementations for all its member functions. An abstract class is the base class from which other classes can be derived. The derived class provides implementations for the member functions that are not implemented in the base class. A derived class that implements all the missing functionality is called a concrete class.

a) landbasedrobot class

Landbasedrobot class servers as the abstract class for landbasedwheeled class and landbasedtracked class. The methods of this class are all virtual methods. When the object for this class is called the constructor initialises the attributes with the default values we have provided. When the object goes out of scope the destructor is called.

All the methods of this class are public and they are:

I. `virtual void GoUp(int x_, int y_)=0;`

Virtual method used by algorithm class to move the robot up to find a path using DFS.

II. `virtual void GoDown(int x_, int y_)=0;`

Virtual method used by algorithm class to move the robot down to find a path using DFS.

III. `virtual void GoLeft(int x_, int y_)=0;`

Virtual method used by algorithm class to move the robot left to find a path using DFS.

IV. `virtual void GoRight(int x_, int y_)=0;`

Virtual method used by algorithm class to move the robot right to find a path using DFS.

V. `virtual void PickUp(std::string& pick)=0;`

Virtual method used to pick up an object when that is the required task

VI. `virtual void Release(std::string& release)=0;`

Virtual method used to release an object when that is the required task

VII. `virtual int get_x() const=0;`

Accessor declared as a virtual method used to access the value of the protected attribute x_. The variable const is used to preserve the value of x_ so that no changes happen when accessing it.

VIII. `virtual int get_y() const=0;`

Accessor declared as a virtual method used to access the value of the protected attribute y_. The variable const is used to preserve the value of y_ so that no changes happen when accessing it.

IX. `virtual char get_direction() const=0;`

Accessor declared as a virtual method used to access the value of the protected attribute direction_. The variable const is used to preserve the value of direction_ so that no changes happen when accessing it.

X. `virtual void set_x(int x)=0;`

Mutator declared as a virtual method used to change the value of the protected attribute x_.

XI. `virtual void set_y(int y)=0;`

Mutator declared as a virtual method used to change the value of the protected attribute y_.

All The attributes of landbasedrobot are protected and are :

I. `std::string name_;`

String variable used to store the name of the robot

II. `int x_;`

Int variable used to store the current x coordinate of the robot

III. `int y_;`

Int variable used to store the current y coordinate of the robot

IV. `double speed_;`

Double variable used to store the speed with which the robot moves

V. `double width_;`

Double variable used to store the width of the robot

VI. `double length_;`

Double variable used to store the length of the robot

VII. `double height_;`

Double variable used to store the height of the robot

VIII. `double capacity_;`

Double variable used to store the capacity of the robot

IX. `char directions_;`

Character variable used to store the current direction of the robot

b) landbasedwheeled class

This is a concrete derived class from LandBasedRobot class and contains the required methods, constructors, destructors and attributes so that the DFS class can make movements to find a valid path from the robot's current position to the goal. This class inherits all the methods and attributes of landbasedrobotclass. These methods are overridden using the virtual technique.

The overridden methods of this class are:

I. `virtual void GoUp(int x_, int y_) override;`

Virtual override method used by algorithm class to move the robot up to find a path using DFS.

II. `virtual void GoDown(int x_, int y_) override;`

Virtual override method used by algorithm class to move the robot down to find a path using DFS.

III. `virtual void GoLeft(int x_, int y_) override;`

Virtual override method used by algorithm class to move the robot left to find a path using DFS.

IV. `virtual void GoRight(int x_, int y_) override;`

Virtual override method used by algorithm class to move the robot right to find a path using DFS.

V. `virtual void PickUp(std::string& pick) override;`

Virtual override method used to pick up an object when that is the required task

VI. `virtual void Release(std::string& release) override;`

Virtual override method used to release an object when that is the required task

VII. `virtual int get_x() const override;`

Accessor declared as a virtual override method used to access the value of the protected attribute x_. The variable const is used to preserve the value of x_ so that no changes happen when accessing it.

VIII. `virtual int get_y() const override;`

Accessor declared as a virtual override method used to access the value of the protected attribute y_. The variable const is used to preserve the value of y_ so that no changes happen when accessing it.

IX. `virtual char get_direction() const override ;`

Accessor declared as a virtual override method used to access the value of the protected attribute direction_. The variable const is used to preserve the value of direction_ so that no changes happen when accessing it.

X. `virtual void set_x(int x) override;`

Mutator declared as a virtual override method used to change the value of the protected attribute x_.

XI. `virtual void set_y(int y) override ;`

Mutator declared as a virtual override method used to change the value of the protected attribute y_.

The attributes of landbasedwheeled class are:

I. `int wheel_number;`

Int variable used to store the number of wheels of the robot

II. `std::shared_ptr<std::string> wheel_type;`

Pointer of string type used to store the wheel type of the robot

When an object is created for this class the constructor initialises the inherited attributes and the attributes of this class with the default values we have provided. The destructor is called when the object goes out of scope. DFS uses the methods of this class to find the path from current start to the goal.

c) landbasedtracked class

This is a concrete derived class from LandBasedRobot class and contains the required methods, constructors, destructors and attributes so that the DFS class can make movements to find a valid path from the robot's current position to the goal. This class inherits all the methods and attributes of landbasedrobotclass. These methods are overridden using the virtual technique.

The overridden methods of this class are:

I. `virtual void GoUp(int x_, int y_) override;`

Virtual override method used by algorithm class to move the robot up to find a path using DFS.

II. `virtual void GoDown(int x_, int y_) override;`

Virtual override method used by algorithm class to move the robot down to find a path using DFS.

III. `virtual void GoLeft(int x_, int y_) override;`

Virtual override method used by algorithm class to move the robot left to find a path using DFS.

IV. `virtual void GoRight(int x_, int y_) override;`

Virtual override method used by algorithm class to move the robot right to find a path using DFS.

V. `virtual void PickUp(std::string& pick) override;`

Virtual override method used to pick up an object when that is the required task

VI. `virtual void Release(std::string& release) override;`

Virtual override method used to release an object when that is the required task

VII. `virtual int get_x() const override;`

Accessor declared as a virtual override method used to access the value of the protected attribute x_. The variable const is used to preserve the value of x_ so that no changes happen when accessing it.

VIII. `virtual int get_y() const override;`

Accessor declared as a virtual override method used to access the value of the protected attribute y_. The variable const is used to preserve the value of y_ so that no changes happen when accessing it.

IX. `virtual char get_direction() const override ;`

Accessor declared as a virtual override method used to access the value of the protected attribute direction_. The variable const is used to preserve the value of direction_ so that no changes happen when accessing it.

X. `virtual void set_x(int x) override;`

Mutator declared as a virtual override method used to change the value of the protected attribute x_.

XI. `virtual void set_y(int y) override ;`

Mutator declared as a virtual override method used to change the value of the protected attribute y_.

The attributes of this class are:

I. std::shared_ptr<std::string> track_type;

Pointer of string type used to store the track type of the robot

When an object is created for this class the constructor initialises the inherited attributes and the attributes of this class with the default values we have provided. The destructor is called when the object goes out of scope. DFS uses the methods of this class to find the path from current start to the goal.

d) api class

The methods used in API class are used to perform tasks on MMS. The functionalities of these methods were provided to us and we have not made any changes to this. All the methods of the API class are called from the Algorithm class.

The methods used in API class are :

I. **static int** mazeWidth();

Static method used to get the width of the maze from MMS and returns the value to algorithm class from where it was called.

II. **static int** mazeHeight();

Static method used to get the height of the maze from MMS and returns the value to algorithm class from where it was called.

III. **static bool** wallFront();

Static method used to check if there is a wall in front of the robot in MMS and returns true to algorithm class if there is a wall, else returns false.

IV. **static bool** wallRight();

Static method used to check if there is a wall on the right side of the robot in MMS and returns true to algorithm class if there is a wall, else returns false.

V. `static bool wallLeft();`

Static method used to check if there is a wall on the left side of the robot in MMS and returns true to algorithm class if there is a wall, else returns false.

VI. `static void moveForward();`

Static method used to move the robot one step in the direction of the robot in MMS.

VII. `static void turnRight();`

Static method used to turn the robot 90 degrees right in MMS.

VIII. `static void turnLeft();`

Static method used to turn the robot 90 degrees left in MMS.

IX. `static void setWall(int x, int y, char direction);`

Static method used to highlight a wall when a wall is detected using wallLeft(),wallRight() or wallFront(). The arguments are the x, y coordinates of the robot and the direction in which the wall is present.

X. `static void clearWall(int x, int y, char direction);`

Static method used to clear a wall (to de-highlight a wall) when necessary. We did not find the need to use this method in our code.

XI. `static void setColor(int x, int y, char color);`

Static method used to assign a particular color to different cells when needed. We used this function to set colors from the start and goal cells.

XII. `static void clearColor(int x, int y);`

Static method used to clear the color from the cell, i.e to remove the color and bring back the default color. We use this to clear the path each time DFS is called when the robot encounters a wall.

XIII. `static void clearAllColor();`

Static method used to clear the color from all the cells in the maze. We did not find the need to use this method in our code.

XIV. `static void setText(int x, int y, const std::string& text);`

Static method used to assign a particular text to different cells when needed. We used this function to name the start and goal cells.

XV. `static void clearText(int x, int y);`

Static method used to clear text from any particular cell. We did not find the need to use this method in our code.

XVI. `static void clearAllText();`

Static method used to clear all the text from the mazel. We did not find the need to use this method in our code.

XVII. `static bool wasReset();`

Static method used to reset the whole method. We did not find the need to use this method in our code.

XVIII. `static void ackReset();`

Static method used to reset the acknowledgement. We did not find the need to use this method in our code.

e) algorithm class

This is the core of the program. All the computation and the movements of the takes place from here. Algorithm class is a derived class of maze. We are creating on object for landbasedwheeled in the main file and we are passing this object as an argument to algorithm class when we call solve() from main. When the program is executed an object is created for landbasedwheeled class. The next step is to create an object for algorithm class. Using this object we call the Clear_Visited_Array() method to make sure the VisitedNodes array elements are '0'. Then we call the ClearStack() method from the main to make sure the solution stack is empty. Then we call SetUp() method to set color and text to the start and goal cells on the maze and also to highlight the walls around the perimeter of the maze. This is done by calling the methods of API class from algorithm class.

Then we call the Solve() method which is the core method. Here we find the current coordinates of the robot from the landbasedwheeel class and perform DFS using the methods of landbasedwheel (GoUp, GoDown, GoLeft, GoRight) while pushing the path into a stack and updating the visited nodes array. If walls are encountered and the robot can not move in any direction then back tracking is done and those nodes are popped from the stack. Once the path is found we send the path to MoveRobot() which moves the robot along the path using the methods of API class. If a wall is encountered in the path, then the wall is updated in the wallarray and solve() method is called and a new path is calculated using the updated wallarray. This process is repeated until the robot reaches the goal. All the methods and attributes of this class are public.

The methods of this class are:

I. `void SetUp();`

This method highlights the walls around the boundary of the maze and highlights and marks the start and goa cells with 'S' and 'G' respectively.

II. `void Clear_Visited_Nodes();`

Changes the value of the elements of VisitedNodes to zero each time the solve method is called.

III. `void Clear_Visited_Array();`

Changes the value of the elements of VisitedArray to zero each time the solve method is called.

IV. `void Print_Visited_Nodes();`

This method is used to print the elements of the VisitedNodes array.

V. `void Solve(std::shared_ptr<fp::landbasedrobot> robot);`

This method is used to find a path from the current location of the robot to the goal.

VI. `void MoveRobot(std::stack<int> path, std::shared_ptr<fp::landbasedrobot> ptr);`

This method is used to move the robot inside the micro mouse simulator

VII. `void log(const std::string& text);`

This method is used to print text in micro mouse simulator

VIII. `void displayNumber(int num);`

This method is used to print numbers in micro mouse simulator

IX. `void PrintStack(std::stack <int> s);`

This method is used to print the contents of the stack

X. `static bool CheckGoal(int x, int y);`

This method is used to check if the current location of the robot is the goal

XI. `int Set_Coordinates(int x, int y);`

This method is used to convert the x and y coordinates into a single number to that it can be pushed into the stack

XII. `std::array<int, 2> Get_Coordinates(int coordinates);`

This method is used to retrieve the x and y coordinates from the number that was pushed into the stack

XIII. `std::stack<int> ReverseStack(std::stack <int> stack);`

This method is used to reverse the order in which the elements are present in the stack.

XIV. `std::stack<int> ClearStack(std::stack <int> stack);`

This method is used to clear the stack each time a new path has to be calculated

XV. `void HighlightPath(std::stack<int> s);`

This method is used to highlight the path the robot has to take to reach the goal inside the micromouse simulator.

The attributes of algorithm class are :

I. `int width;`

Stores the width of the maze

II. `int height;`

Stores the height of the maze

III. `char dir;`

Stores the direction the robot is facing

IV. `char direc;`

Temporary variable to store the direction

V. `int x;`

Stores the current x coordinate of the robot in the maze

VI. `int y;`

Stores the current y coordinate of the robot in the maze

VII. `int coordinate;`

Stores the coordinate containing both x and y

VIII. `int m;`

Stores the future x coordinate of the robot in the maze

IX. `int n;`

Stores the future y coordinate of the robot in the maze

X. `int p;`

Temp coordinate variable

XI. `int NewCoordinate;`

Variable used to store the popped off value from stack

XII. `std::array <int,2> point;`

Stores the x and y coordinate

XIII. `std::array<std::array<int, 16>, 16> VisitedNodes;`

Boolean array that keeps track of x and y coordinates visited by the robot while planning

XIV. `std::stack<int> Stack;`

Used to store the path from current node to the goal node

XV. `std::stack<int> Solution;`

Stores the Final path from start to goal node

XVI. `std::stack <int> temp;`

Temporary stack used when stack size is 1

XVII. `std::array<std::array<int, 16>, 16> VisitedArray;`

Integer array that keeps track of x and y coordinates visited by the robot while moving in the maze

f) Maze class

This class is the base class for Algorithm class. All the methods and attributes of this maze class are public. The attributes of this class stores the information from the maze loaded from the simulator. The methods of this class helps the program to set and reset the wall information for two adjacent cells in the maze. The overall final path from the start to the goal location is decided based on the wall information stored in the WallArray array attribute of this class.

The methods used in maze class are :

I. `std::array<std::array<int, 256>, 256> InitializeMaze();`

Initializes the elements of WallArray with 0

II. `std::array<std::array<int, 256>, 256> SetWall(int x, int y);`

This method is used to set the coordinates of WallArray as true each time a wall is found so that DFS can avoid these points while finding the path to goal.

III. `int GetDimension();`

This methods returns the dimensions of the WallArray

IV. `bool CheckWalls(int x, int y);`

This method is used to check if there is a wall between two coordinates. The method returns true if a wall is found

V. `int` SetCoordinates(`int` x, `int` y);

Sets WallArray index to true if there is a wall between two coordinates in the maze

The attributes of algorithm class are :

`int` coordinate{};

Variable which stores the coordinate value of the current x and y value

std::array<std::array<`int`, 256>, 256> WallArray{};

Array to store all the Wall information about the maze in MMS

g) Direction Struct

This is not a class. This is a type of data structure in c++ which is slightly similar to classes. We are using this to store the different directions.

The attributes of direction are :

`char` NORTH

`char` EAST

`char` SOUTH

`char` WEST

Contributions

1) Pradeep Gopal
    - Did a major part of the coding and debugging.
    - Used his logic to solve DFS
    - Helped to clean up the code and comment
    - Provided logic help when needed
2) Markose Jacob
    - Figured out how to use the methods of API class
    - Helped to code the way the robot follows the path in the maze
    - Helped in coding
    - Did major part of the report
    - Implemented struct data structure to store direction
    - Provided logic help when needed
3) Srikumar Muralidharan
    - Gave us multiple solutions whenever we needed logic to execute any parts that we are stuck on.
    - Helped in fixing bugs.
    - Tried the concept of Visited Array's for robot movement to remove redundancy
    - Did major part of the presentation
    - Helped to clean up the code and comment
    - Provided logic help when needed
4) Rohit Thakur
    - Helped in coding
    - Did major part of the documentation
    - Provided logic help when needed
    - Significantly contributed to make the presentation.

Room for Improvement

Fewer lines of code:

We feel that we did our best to implement the code in as few lines of code as possible using the concepts of Polymorphism, but with more in-depth understanding of C++ functionalities, data structure and algorithm knowledge we think it is possible to write this program more efficiently. As of now, we have implemented functions inside classes and have also implemented the concept of inheritance, where we are taking the information from a base class, LandBasedRobot and are implementing two derived classes LandBasedTracked and LandBasedWheel. However, our Polymorphism strength is not too high so we could not very efficiently implement reusability of code to the maximum extent. However, with further practice, we are confident that we can improve efficiency.

Convoluted Condition Checks:

Some of the conditions we are checking might not be very simple for a layman to quickly understand while going through our code, unless he takes time to analyze what is happening at a particular iteration for a particular maze. We are sticking to the given conditions for moving in the planning environment, however, we are introducing 2 different types of visited arrays, one each for when the robot's motion is planned and the other for when the robot is in motion. We are doing this to stop redundancy while the robot is in motion, as it goes all the way to the start node and then comes back to the further nodes, rather than exploring the newer steps, when it is in motion.

Unexpected/Unreliable Behaviour:

We are getting unexpected results for certain mazes. This generally happens because of a case of unsolvability, when the path taken by the object seems to overlap with its own, the robot gets stuck in an infinite loop as it is unable to move forward and choose another viable path. This is most certainly because of a bug that we are unable to

identify properly and should be fixed. Our code is working reliably for most of the mazes, but sometimes due to the constraints set by Visited Array attribute, we are facing a few problems with the solving part.

Conclusion

We were successfully able to move the robot through the maze till the goal for any maze in the micro mouse simulator. The robot calculates the path using DFS everytime a wall is encountered. The robot always tries to move down then move right then move up and then move left. The path was successfully calculated every time solve() is called. The path to the goal is displayed every time a new path is generated. Since the walls are unknown and since the robot always tries to move down first, it takes a while to reach the goal but the robot always finds its path to the goal.