FINAL PROJECT REPORT

Path Planning algorithm for a land based robot

Group 7 - Jonathon Kreinbrink , Esther Idu Ojokojo ,
Hari Krishna Prannoy Namala

Introduction

The aim of the project is to develop a path planning algorithm for a range of robots. The path needs to be planned from the left-bottom corner of the maze to the center of the maze.

The path planning algorithm used in this project is the Depth First Search (DFS) algorithm. Depth First search 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. An illustration is shown in Figure 1 where the green line is the path generated by the algorithm for the start position A and goal positions N and J.

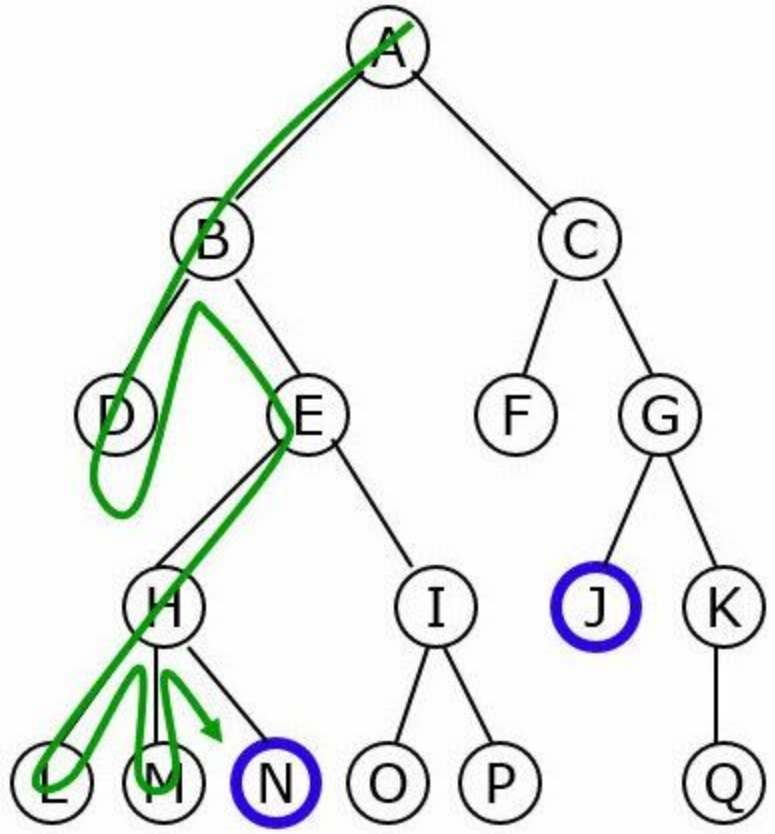

Figure 1. A tree representation showing implementation of DFS

Overview of the Approach

Classes:

Provided Class, API:

The API class was provided as a means to interact with the MicroMouseSimulator program. This class contains functions that permit the modification and display movement of the mouse within mazes. The functions included in this class are shown below:

```
int mazeWidth();
int mazeHeight();

bool wallFront();
bool wallRight();
bool wallLeft();

void moveForward(int distance = 1); // can result in "crash"
void turnRight();
void turnLeft();

void setWall(int x, int y, char direction);
void clearWall(int x, int y, char direction);

void setColor(int x, int y, char color);
void clearColor(int x, int y);
void clearAllColor();

void setText(int x, int y, string text);
void clearText(int x, int y);
void clearAllText();

bool wasReset();
void ackReset();

int/float getStat(string stat);
```

Figure 2. Classes programmed into API class.

In this project, all functions used can be grouped into three main categories, wall functions, movement functions, and UI functions.

Wall Functions:

These functions are frequently called by the classes developed in this project. These functions are logically grouped by purpose as shown in Figure 2. The wallFront, wallRight(), and wallLeft() functions return booleans based on whether or not a wall

exists in the respective direction of the mouse in the program. When one of these three functions return true, the setWall() function can be used to tell the micromouse simulator to show a wall has been discovered.

Movement Functions:

The primary and only function that moves the mouse along the track is moveForward(), this function returns a “crash” and the program fails if attempting to move into a wall of the maze. Having the program check surrounding walls before moving is critical to avoid crashing and failing the program. turnLeft() and turnRight() control whether or not the mouse turns left or right, respectively.

User Interface Functions:

User interface functions are used to visually alter text and color of the micromouse simulator. The main functions used are the setColor() and clearAllColor(), these are used to color the goal cells, color the DFS generated path cells, and clear all colors between runs.

Developed Classes:

Robot Classes:

The robot class is developed into three separate classes, LandBasedRobot, LandBasedWheeled, and LandBasedTracked. LandBasedRobot is the root class with its attributes set to protected, and functions set to public. These functions include the basic turning and movement functions as well as getters to access the current position and direction of the robot.

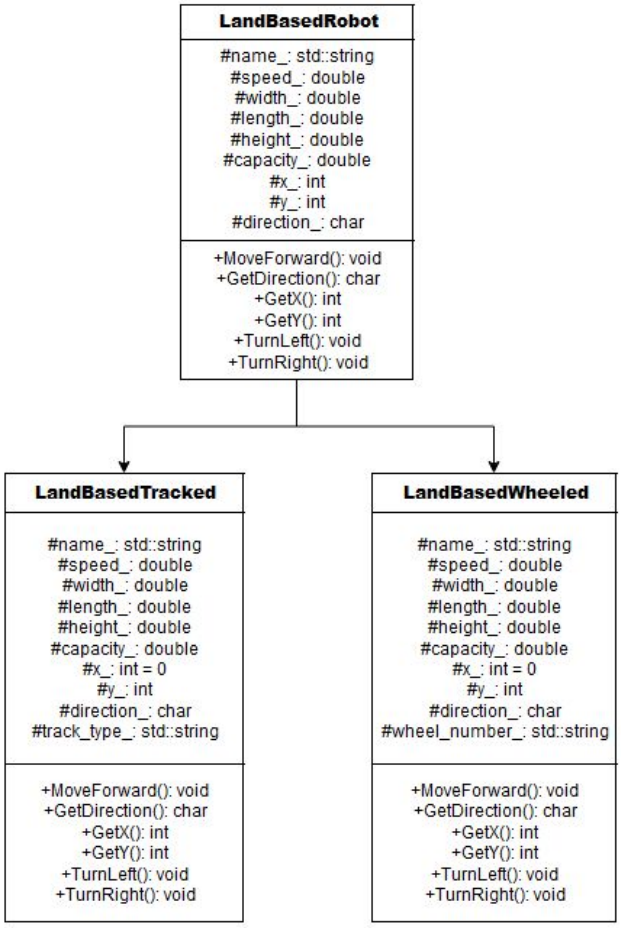

Figure 3: Inheritance chain of Robot Classes.

As seen in Figure 3, the inheritance chain between the root class and derived classes can be seen, with only the “track_type_” and “wheel_number_” attributes being added to the derived classes. The movement functions both update the position of the robot’s position and direction attributes, but also calls the respective function in the API class to move the Mouse in the Micromousesimulator.

Maze:

The purpose of the Maze class is to translate and store data output from the API class into a local class, to allow manipulation and easy storing of data.

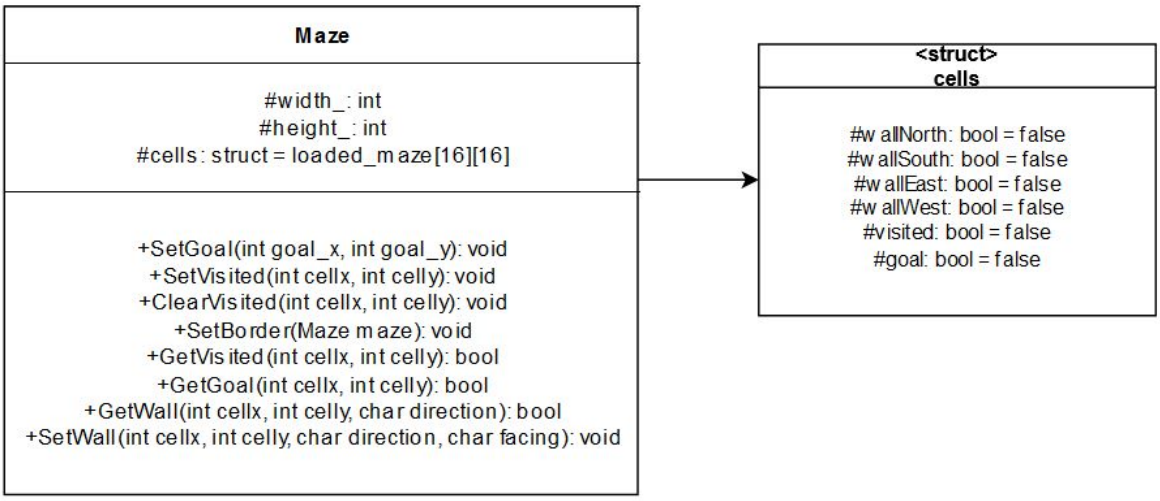

Figure 4. Methods and attributes of Maze class.

The maze class's attributes are primarily stored into a default 16 x 16 data struct named "cells". This data struct mimics the size of a classic maze and acts as a local copy of all data pulled from API functions. Using the getters and setters within the Maze's methods, one can either enter that data into the maze to be used later, or pull data from the maze to determine whether movements can be done, or searching algorithms have visited cells already. The setters in this class both update the loaded_maze data struct and act as callers to the API function.

Algorithm:

The algorithm is created as a class object with a singular function "void Algorithm::Solve". This function can be broken up into four main phases: Creation of objects + setup, Goal Loop, DFS algorithm, Movement algorithm. See Figure 5 to see a flowchart showing the separate while loops and flow of the function.

Algorithm Logic Diagram

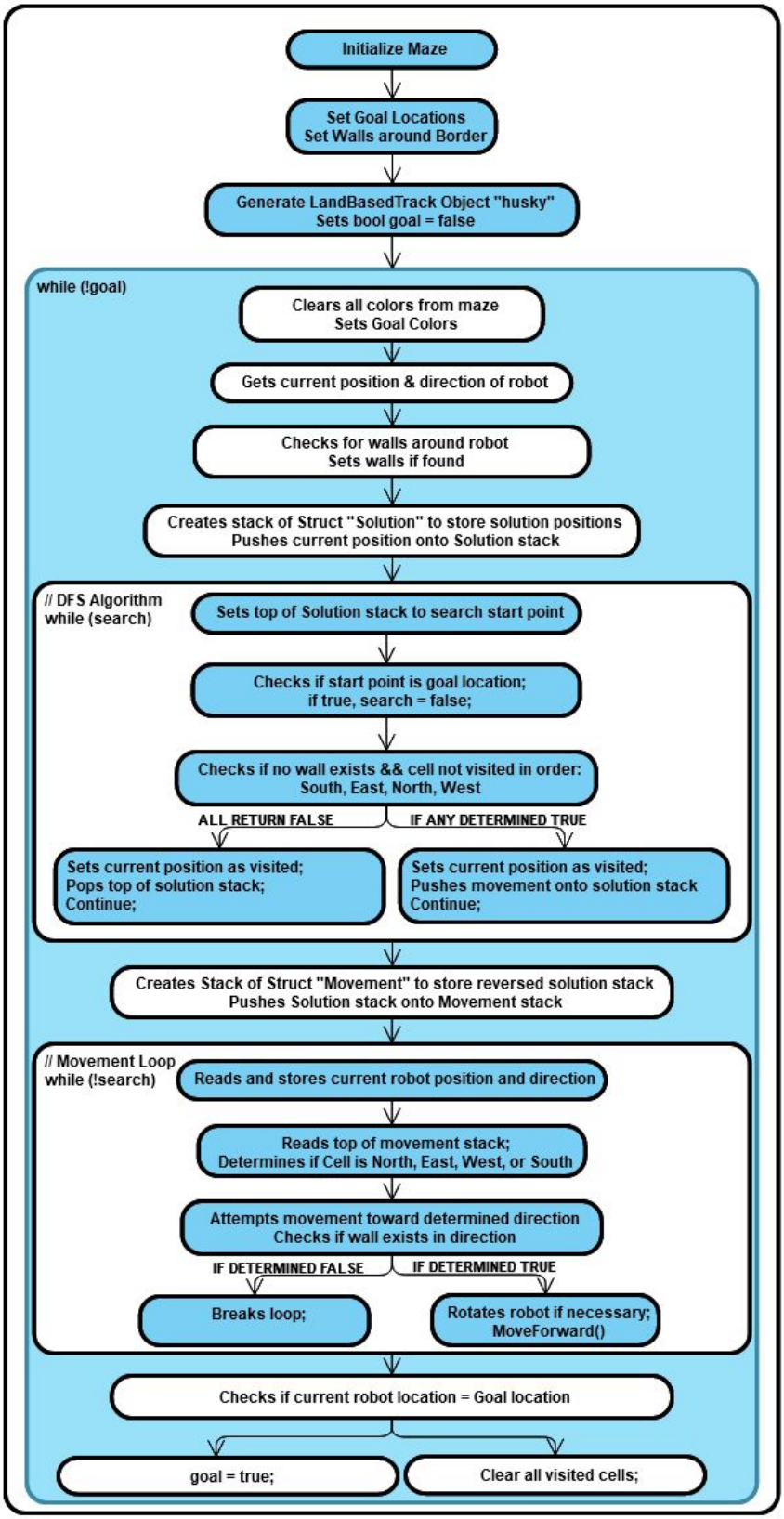

Figure 5. Flowchart of the Algorithm::Solve function.

Creation of Object + Setup:

The initial stage of the algorithm creates a maze object from class Maze, sets goal locations and border walls within the maze, as well as creates an object “husky” out of the LandBasedTrack class. This stage also creates a bool goal and sets it to false.

Goal Loop; Prior to DFS algorithm:

The goal loop uses the boolean “goal” as it’s condition, as long as this returns false, the robot continues searching and moving. This goal loop encompasses both the DFS algorithm and the movement algorithm. The first stage of the goal loop is to clear the previous path from the system, set the goal cells to the color red, and pull the current position and direction of the robot in the maze. Once the algorithm retrieves the location and orientation of the robot using the robot’s getter functions, it checks the surrounding walls and updates the maze and Micromouse simulator if walls are found. Before proceeding into the DFS algorithm and movement algorithm, a data structure “Solution” , consisting of two ints, is created. Subsequently, a stack is generated utilizing the solution struct, this stack will be utilized in the DFS algorithm to store the path the robot will need to take to reach the goal cell.

DFS Algorithm:

A while loop is created for the DFS algorithm, with multiple conditions that utilize the continue statement after adding the next solution cell onto the stack. At the beginning of the DFS algorithm, the program sets the current position to begin the search equal to the position at the top of the solution stack. It then checks if the current position is a goal cell, if this returns true, the function sets the bool “search” equal to false to exit the loop, as it has found a path all the way to the goal cell. The robot then searches in the following directions, in order: South, East, North, West. Each time the algorithm searches in these directions, it checks if no wall exists between the current cell and the proposed cell, and checks if the algorithm has not visited that cell. If both of these return true, the algorithm sets the current cell to visited, pushes the proposed position onto the stack, and restarts the DFS algorithm to continue searching.

One major thing that is necessary to allow backtracking is if a scenario arises where the mouse has searched all directions, and no viable path is found, the top of the

solution stack is popped off, and the searching algorithm continues from the previously visited cell. This allows the mouse to essentially backtrack if it reaches a dead end.

Goal Loop, Between DFS & Movement Algorithm:

Before entering the movement algorithm, we need to reverse the stack generated by the DFS algorithm, as the first entry into the stack is the first required movement. Another stack of data struct “Solution” is created, named “movement” and a while loop is made to enter the top of the solution stack into the movement stack, pop the top of the solution stack, and reiterate until the solution stack is empty.

Movement Loop:

The movement loop begins by reading the robots current direction and position, and storing this data for comparisons between the movement stack. The algorithm then compares the x and y values from the top of the movement stack to the robot’s current position to determine the necessary direction the robot needs to move to. The algorithm then checks the walls around the robot and updates the maze class and micromouse UI. The robot then verifies there is no wall between its current cell and next movement, if a wall exists the loop breaks. If a path is deemed available, the robot turns if necessary then calls the moveForward() function, and repeats the loop until the movement stack is empty, where it should be located at one of the goal cells.

Goal Loop; After Movement Loop:

Finally, after the movement loop, the current robot’s position is compared to the goal cells. If the robot is determined to be located on a goal position the boolean goal is set to true, and the program outputs that the mouse has reached the goal. This concludes the function.

On the other hand, if the mouse is determined to not be located on a goal location, this means the movement loop was unable to reach the goal cell due to a wall. The loop then sets all the cells back to non-visited, and the goal loop restarts to generate another path.

Main:

The main source file simply makes the pointer between the base class and derived class shared, creates an algo object out of the Algorithm class, and calls the algo.Solve() function.

```cpp
/** \file main.cpp
 * FP -Group#7
 */

#include ...

using namespace fp;

    int main() {
        std::shared_ptr<fp::LandBasedRobot> wheeled = std::make_shared<fp::LandBasedWheeled>("husky");
        fp::Algorithm algo; // Creates algo object from Algorithm class
        algo.Solve(); // Calls solve function
        return 0;
    }
```

Figure 6. Main.cpp source file

Challenges Faced

In this section, the challenges faced during the execution of the project are listed below:

- Using an external UI has made debugging a challenge.
- While testing out the code of the robot in the maze, the robot was not moving forward and not able to recognize walls.
- There were issues with stacking visited cells.
- Orienting which attributes and methods should belong to each class and properly setting them up to access available attributes.
- After the goal was reached, a run output was observed which says "**stack smashing detected : <unknown> terminated**". This is an unresolved error which does not obstruct the algorithm and a solution could not be found.

Contributions

Although there was no definite sharing of work among the team, a broad division of work is listed in below.

- Most of the groundwork was laid by Esther.
- Jonathon has made major headway towards the algorithm.
- Prannoy was mostly involved with the report and presentation.

Conclusions

The currently implemented DFS algorithm is not the best available algorithm to obtain the path. Since all the cells in the branch are visited in DFS, the computation time is higher. Use of advanced path-planning algorithms can be implemented. One such example is A* search algorithm. These advanced algorithms will improve the efficiency of the program and the path can be obtained with completeness and optimal efficiency. This is highly important for bigger mazes as the computation time will increase rapidly with increase in the size of the maze due to the obvious reason of increased size of each branch.

Also, the DFS algorithm does not guarantee a solution in all the cases. In case of an infinite number of cells on a branch, DFS will continue to go to the end of the branch resulting in an endless loop. For DFS the path which it finds simply depends on the order in which it explores the successor nodes, which may or may not yield a shortest path.

Finally,the maze can be better represented using adjacency lists. An adjacency list is a collection of unordered lists in which each element contains the locations of neighbouring cells for one cell of the maze. There are many successful implementations of Dijkstra's *Algorithm* (which is the algorithm based on which A* search algorithm was developed) with the help of adjacency lists for path planning applications.