Self Correcting Maze Solving AI for Hamster-S Robot

Caleb Millard

613362

Dr. Andrew Park

CMPT 380

April 14, 2023

Competition Research:

There are a lot of micro-mouse competitions, and every single one that I have found allows you to produce your own personal micro-mouse. I do think this would allow for greater competition, provided we had time. Almost all competitions use very similar walls to what we use in our competition. All of the code must be self-contained, so you are not allowed to touch the code or your computer at any point during the run or while your mouse is on the maze; along with this, you are not allowed to edit your code after the maze has been revealed.

Constraints in most Competitions:

Quoted directly from the IEEE competition rules in 2020:

"3.1. Fabrication

The MicroMouse robot submitted by a team must be designed and built from scratch.

3.2. Self Containment

The MicroMouse robot shall be self-contained (no remote controls). The robot shall not use an energy source employing a combustion process.

3.3. Dislodged Parts

The MicroMouse robot shall not separate from any part of itself whilst navigating the maze. To complete the maze, the robot in its entirety must enter the center of the Maze

.

3.4. Method of Movement

The MicroMouse robot shall not jump over, fly over, climb, scratch, cut, burn, mark, damage, or destroy the walls of the maze.

3.5. Micromouse Size

The MicroMouse robot shall not be larger either in length or in width than 25 centimeters. The dimensions of a MicroMouse robot that changes its geometry shall not be greater than 25cm x 25cm (length and width, respectively). There are no restrictions

on the height of the robot.

3.6. Inspection
All MicroMouse robots are subject to inspection before starting their competition to ensure they are within the specifications outlined by these rules and that they do not pose potential safety hazards.

3.7. Rules Violation
Any violation of these rules will constitute immediate disqualification from the contest and ineligibility for any associated prizes."[5]

Scoring:

IEEE:
"6.2. Scoring First place goes to the team with the shortest official time (without the team touching its MicroMouse robot during runs). Second prize to the team with the next shortest time, and so on. Teams with the MicroMouse robot that does not enter the center square will be ranked by the judges based on the following criteria: ● How close the robot gets to the destination square without being touched. ● Evidence that the mouse knows where it is relative to the destination square. If, on occasion, the robot becomes immobilized in a corner or on a wall, the team may manually intervene to correct the problem (with care not to modify the mouse's intended direction of movement. The frequency of such corrections will be considered by the judges while scoring.

Generally, in most competitions, a penalty is applied to the time of the fastest run based on the speed of previous runs, even if they are mapping runs.

There is also a penalty for touching your robot in most competitions, allowing you to clean its wheels or for any other purpose."[5]

Conclusion of Research on Competitions:

I designed my robot according to the rules of the set that you are unable to preprogram any solution to the maze aswell and you may return to the starting point of the maze to restart a run. Using this, I based my algorithms accordingly, as the time in the start square does not matter in terms of run time or overall time spent. The competitions are pretty free form allowing for a lot of innovation in your design and build.

Possible search algorithms:

Depth-first:
I took one look at this type of search, and I understood that it could return a nonefficient solution or path to the maze and decided against it

Left Hand:
Inefficient and has a chance, based on maze design, that it may not work as a solve

Right hand:
Similar as left-hand solve, it may not find the goal base on how the maze is designed

Dijkstra's Algorithm:
This is a weighted depth-first essentially, which is an excellent solution for maze solving, and I heavily considered using this algorithm as it would most likely find me the most efficient path if I assigned weights to the different nodes in my maze findings[]

A-star:
A star would be a very good solution, but without knowing where the goal is or the structure of the maze, this could prove difficult; this was another algorithm I heavily considered since a lot of micro-mouse teams in collegiate competitions use this algorithm. It does require significantly more testing and is more challenging to implement than some other more simple algorithms and

based on the time given for this assignment, I chose to stick with more simple algorithms, and if I had time, then I would adapt and change my algorithm later. And even if I could apply heuristics by knowing we travel to the center square, this would be just as efficient as other searches

Kalman Filter:

This algorithm would be very useful as it requires predictions and probability usage to find the most intuitive way of solving the maze in an uninformed search kind of way, but this also requires in-depth research and development to make this work, so I pulled some elements of this algorithm into what I implemented[1]

Breadth-First:

A breadth-first search is guaranteed to find the most efficient path to the goal state, but it is very inefficient

FloodSearch:

flood the maze with a search-like breadth first but all simultaneously and see where first hits the goal. To add weight to this, it essentially adds one to the search each time and then will return the most efficient path in a short amount of time at the cost of efficient space complexity

Search efficiencies:

Algorithm	**Time Complexity(average)**	**Why is this bad or good**
Dijkstra's (uninformed) Dijkstra's (informed)	O(V^2) O((V + E) log V)	This is a semi-efficient algorithm that is not the most efficient because it searches depth and may not return the most efficient path since the data is not sorted when unsearched.

Left-Hand Search Solve	O(n)	Complexity is the size of the maze in terms of cells
Right-Hand Search Solve	O(n)	Complexity is the size of the maze in terms of cells
A-star	O(b^d*h)	Is efficient assuming you can use heuristics though this can
Breadth-First Search	O(V + E)	Very efficient and finds the best possible path
Weighted Flood Search	O(V + E)	Very efficient and finds the best possible path
Depth First Search	O(V + E)	Very efficient and sometimes finds the best possible path, but not often

[1]

Variable Table:

h is the heuristic value of the starting node in the maze.

V represents the number of cells in the graph.

E represents the number of paths in the graph.

n represents the size of the maze in cells.

b represents the branching factor(the number of possible deviations from the correct path).

d represents the longest potential path.

Reason for choosing flood search for uninformed search:

I chose to use a weighted flood search to find the most efficient way to the goal; since we know that the goal would be in the center of the maze, we would be able to move in the direction of the middle and most likely reach it faster than if we explored the entirety of the maze initially.

So with this, I do a one-deep weighted flood search in my program that scans all adjacent squares and decides in which direction to go based on which one is closer to the goal. If it finally reaches the goal, then the weighted portion will be dropped, and it will continue to find every unexplored space in the maze.

Once the goal has been found and the maze has been thoroughly and completely searched, the lights on the robot will turn green, wait ten seconds then commence the solved maze most efficiently. If there is no viable path to the goal, the lights will turn red and then give a low buzz.

A study done by Cornell University explains that flood searches are one of the best algorithms to use on weighted node graphs, and since I apply a weight to the nodes in my matrix, this makes it one of the best choices for my uninformed search.

I also did some research into the Kalman filter, which is used by a lot of micro mouse teams, and implemented some of the predictions that they would put into their algorithms to make my weighted flood search more efficient; I did this by prioritizing moves that would put the mouse closest to the goal and be able to predict where might lead to a dead end based on walls.

Reasoning for choosing a breadth-first solving algorithm after the maze has been searched:

I could have chosen a depth-first search as it is significantly faster in terms of finding a path in a giant maze. Still, since our maze is relatively small, I realized we do not need the most efficient algorithm to solve this puzzle aswell, and using a depth-first search does not guarantee the fastest solution to the maze.[1]

In many micro mouse competitions, there is no need for speed until the end and final solve of the maze; in this reasoning, since the time is unaffected while you are calculating the path, even if you choose an inefficient algorithm, as long as it can find the most efficient path to take to the goal, then all it comes down to is the efficiency of movement for your mouse. [5]

In this competition, the mice were the most challenging thing to overcome regarding what was limiting us. They were inaccurate regarding both sensors and moving in a straight line, and when turning, they would turn anywhere between 85 and 95 degrees when told to turn 90. So, in the end, my thought process behind choosing a breadth-first search is that since there is no rugged terrain, it would select the shortest path to get to the goal.

I could have chosen to use the Djyikstras algorithm for this, but it is essentially the same algorithm. Still, with weights, and in this situation, I found it would be significantly more time-wise to use the simpler algorithm to work on the issue preventing me from completing the task.

Experimental results:

1= wall

0 = path

2 = unexplored

3 = path to goal

4 = goal

Initial map before any search:

[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1]

[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1]

[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1]

[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1]

[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1]

[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1]

[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1]

[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1]

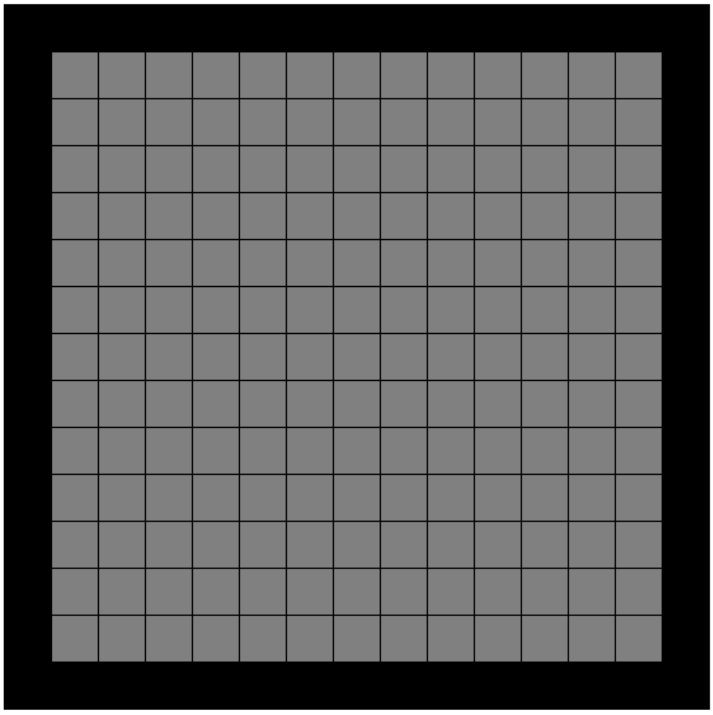
[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1]

[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1]

[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1]

[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1]

[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]



The experimental result after initial mapping using the flood search:

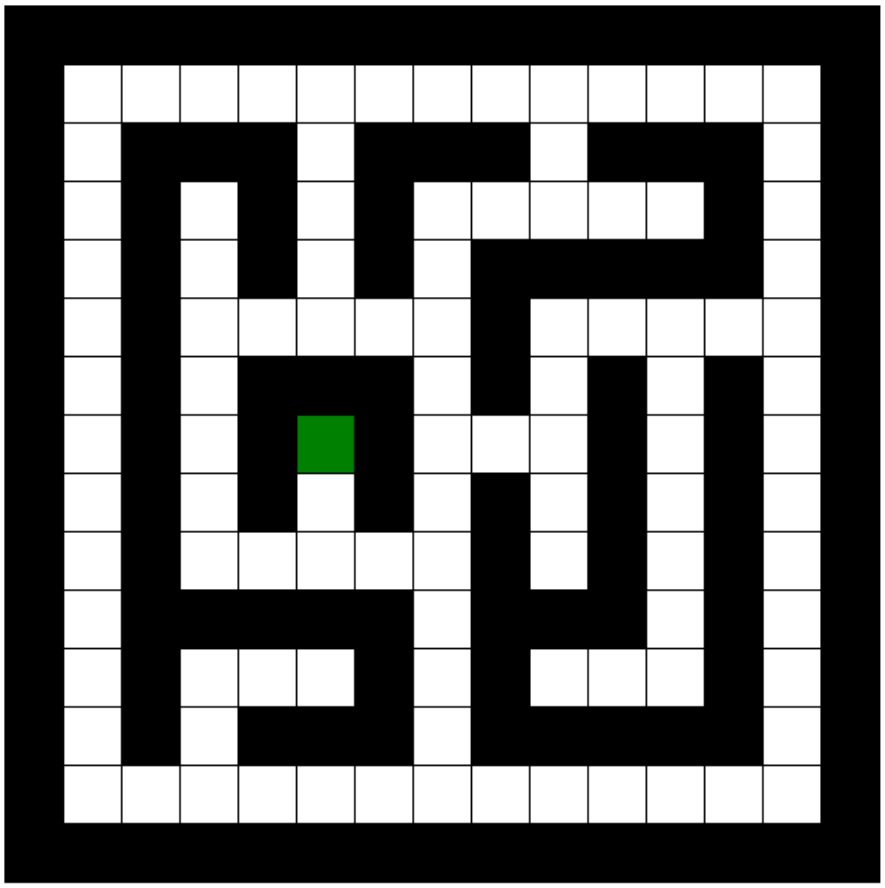
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
[1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1]
[1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1]
[1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1]
[1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1]
[1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1]
[1, 0, 1, 0, 1, 4, 1, 0, 0, 0, 1, 0, 1, 0, 1]
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
[1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1]
[1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1]

[1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1]

[1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1]

[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]

[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]



After the breadth-first algorithm finds the most efficient route:

3 = path

4 = goal

[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]

[1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1]

[1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1]

[1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1]

[1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1]

[1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1]

[1, 0, 1, 0, 1, 4, 1, 0, 0, 0, 1, 0, 1, 0, 1]

[1, 0, 1, 0, 1, 3, 1, 0, 1, 0, 1, 0, 1, 0, 1]

[1, 0, 1, 0, 0, 3, 3, 3, 1, 0, 1, 0, 1, 0, 1]

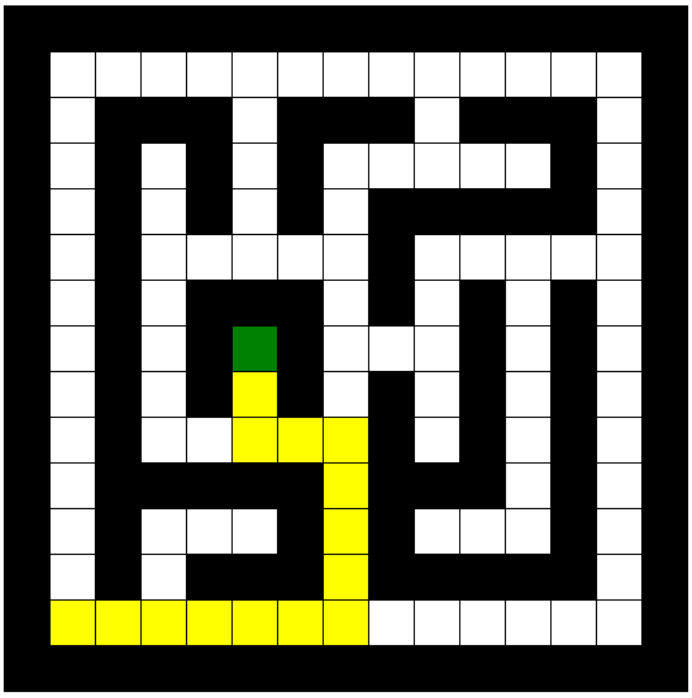
[1, 0, 1, 1, 1, 1, 1, 3, 1, 1, 1, 0, 1, 0, 1]

[1, 0, 1, 0, 0, 0, 1, 3, 1, 0, 0, 0, 1, 0, 1]

[1, 0, 1, 0, 1, 1, 1, 3, 1, 1, 1, 1, 1, 0, 1]

[1, 3, 3, 3, 3, 3, 3, 3, 0, 0, 0, 0, 0, 0, 1]

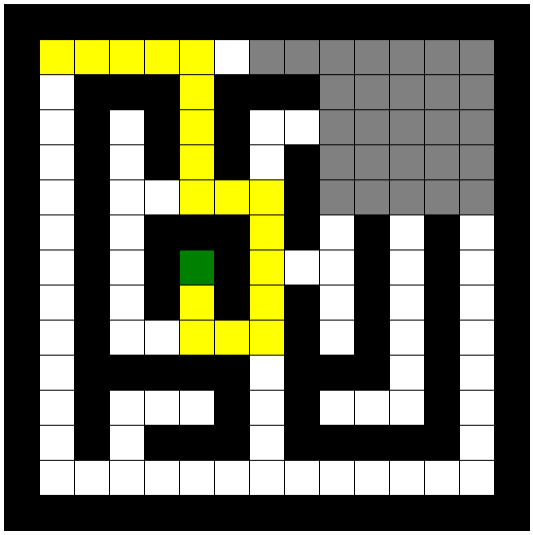
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]



In this instance the path returns the following values [(13, 1), (13, 3), (13, 5), (13, 7), (11, 7), (9, 7), (9, 5), (7, 5)]

But has tested all of the following nodes: {(3, 1), (5, 1), (9, 5), (11, 5), (13, 5), (13, 11), (7, 1), (7, 7), (9, 1), (9, 7), (11, 1), (11, 7), (13, 1), (13, 7), (11, 13), (13, 13), (9, 3), (1, 1), (11, 3), (13, 3), (13, 9)}

If forced to solve the algorithm early it will still function and achieve the goal since using a breadth first search does not require knowledge of the entire maze to run. Since it can see all the cells around the goal it can determine from cell (1,1) this is the most efficient path.



Screenshots of code with explanation:

```
if direction == "right":
    if orientation == 0:
        turnRight()
        OrientSelfFromWall()
        checker()
        if (
            map[position[0]][position[1] + 1] == 1
            and map[position[0]][position[1] - 1] == 1
            and (orientation == 0 or 3)
            or map[position[0] + 1][position[1]] == 1
            and map[position[0] - 1][position[1]] == 1
            and (orientation == 1 or 2)
        ):
            turnCorrecter()
            calibrate()
        moveForward()
        checkWallCurrentNodeUp()
        checkWallCurrentNodeRight()
        checker()
```

An example of my movement function is if it has received the command to move to the right and if it is facing up, then it will turn right and orient itself; if its orientation is skewed, it will calibrate and correct, then move forward and scan the next cell.

```
# this will check behind the bot and then ram the bot into it and then move forward to reset distance
def OrientSelf():
    global orientation
    global position
    if orientation == 0:
        print("fixing down")
        print((map[position[0] + 1][position[1]]))
        if (map[position[0] + 1][position[1]]) == 1:
            hamster.wheels(-70)
            wait(500)
            hamster.wheels(30)
            wait(600)
    if orientation == 1:
        print("fixing left")
        print((map[position[0]][position[1] + 1]))
        if (map[position[0]][position[1] + 1]) == 1:
            hamster.wheels(-70)
```

This method uses the wall behind the bot and backlash the bot into it and then moves forward, allowing the bot to recenter itself in the cell.

```
# this will return a solved path to the goal from th
def solver():
    hamster.leds(Hamster.COLOR_NAME_GREEN)
    wait(10000)
    global orientation
    orientation = 1
    position = (5, 9)
    solve = find_Home()
    if solve != []:
        print(solve)
    for path in solve:
        if path[0] > position[0]:
            move_func_Solve("down")
        elif path[0] < position[0]:
            move_func_Solve("up")
        elif path[1] > position[1]:
            move_func_Solve("right")
        elif path[1] < position[1]:
            move_func_Solve("left")
    if position == (9, 5):
        victory()
    else:
        solver()

```

Solver, after the maze has been mapped, will then use a breadth-first search and find the most efficient path to the goal from the start of the maze; it will then execute the path by sending directions to the move solve function; if it fails, it will then try and solve again

```
# draW a map of the maze
def Draw():
    for row in map:
        for element in row:
            if element == 0:  # white square
                turtle.fillcolor("white")
            elif element == 1:  # black square
                turtle.fillcolor("black")
            elif element == 4:  # green target square
                turtle.fillcolor("green")
            elif element == 3:  # path
                turtle.fillcolor("yellow")
            else:  # grey square
                turtle.fillcolor("grey")
            turtle.pendown()
            turtle.begin_fill()
            for i in range(4):
                turtle.forward(square_size)
                turtle.right(90)
            turtle.end_fill()
            turtle.penup()
            turtle.setposition(turtle.xcor() + square_size, turtle.ycor())
        turtle.setposition(-200, turtle.ycor() - square_size)

    turtle.done()
```

This draws the excellent maps of the maze as seen earlier

```python
def find_Path():
    global position
    start = tuple(position)
    queue = [(start[0], start[1], [])]
    visited = set()

    while queue:
        row, col, path = queue.pop(0)

        if map[row][col] == 2:  # Check if the current cell matches the value 2
            path = path
            return path

        if (row, col) not in visited:
            visited.add((row, col))

            # Generate next possible moves
            moves = [(0, 2), (0, -2), (2, 0), (-2, 0)]
            for move in moves:
                new_row = row + move[0]
                new_col = col + move[1]

                # Check if the move is valid and does not jump over walls
                if (
                    is_valid_move(map, new_row, new_col)
                    and (move[0] == 0 or map[row + int(move[0] / 2)][col] != 1)
                    and (move[1] == 0 or map[row][col + int(move[1] / 2)] != 1)
                ):
                    queue.append((new_row, new_col, path + [(new_row, new_col)]))

    return None
```

This is a breadth-first search to find the nearest unexplored cell in the maze and then will return the path to the nearest unexplored cell in the form of coordinates describing the next cell needed.

```
def moveForward(wheelspeed=87):
    setPositionForward()
    hamster.stop()
    wait(100)
    print("moving forward")
    marginOfErrorchecker()
    for i in range(78):
        checker()

        if R >= 38:
            diff = R - 40
            hamster.wheels((wheelspeed) - (diff * 0.6), (wheelspeed) + (diff * 0.6))
            wait(10)
        elif L >= 38:
            diff = L - 40
            hamster.wheels((wheelspeed) + (diff * 0.6), (wheelspeed) - (diff * 0.6))
            wait(10)
        elif R <= 38 and R >= 25:
            diff = R - 40
            hamster.wheels((wheelspeed) - (diff * 0.6), (wheelspeed) + (diff * 0.6))
            wait(10)
        elif L <= 38 and L >= 25:
            diff = L - 40
            hamster.wheels((wheelspeed) + (diff * 0.6), (wheelspeed) - (diff * 0.6))
            wait(10)

        else:
            hamster.wheels(wheelspeed)

            wait(10)
```

This is one of the most impressive functions I programmed, which is a weighted driving algorithm. This will correct the robot as it is moving forward, allowing itself to align in the center of the aisle of the maze. If there are not two walls on either side, it will select the closest wall and align itself with that wall making sure it is 40 units away from that wall which puts it in the center of the cell. Along with this, it also checks every ten milliseconds allowing for fast and almost unnoticeable changes while maintaining a correct straight path

```python
def check_adjacent_2d_array(arr, row, col):
    rows = len(arr)
    cols = len(arr[0])

    # Check top
    if row > 0 and arr[row - 1][col] == 0:
        if row > 1 and arr[row - 2][col] == 2:
            return "up"

    # Check bottom
    if row < rows - 1 and arr[row + 1][col] == 0:
        if row < rows - 2 and arr[row + 2][col] == 2:
            return "down"

    # Check left
    if col > 0 and arr[row][col - 1] == 0:
        if col > 1 and arr[row][col - 2] == 2:
            return "left"

    # Check right
    if col < cols - 1 and arr[row][col + 1] == 0:
        if col < cols - 2 and arr[row][col + 2] == 2:
            return "right"

    return None
```

This is my flood search without the weight; the weight is added when the array is checked. It checks all nearby nodes and then selects which one it will travel to.

```
#calibrates and centers the robot by aligning it
def calibrate():
    print("calibrating")
    hamster.wheels(-30)
    wait(500)
    checker()
    if L > 43:
        turnRight()
        OrientSelfFromWall()
        turnLeft()
    elif R > 42:
        turnLeft()
        OrientSelfFromWall()
        turnRight()
    checker()
    hamster.wheels(30)
    wait(300)
    hamster.stop()
    print("done calibration")
```

This calibration method will allow the robot to find a nearby wall and then center itself in the cell by rotating and reversing into the wall until it is flush with the wall, then goes forward and rotates back to its original orientation. This will return the robot to the center of the cell.

```
def instantiate():
    global orientation
    orientation = 1
    global position
    position = [5, 9]
    global map

    map = [[2 for _ in range(15)] for _ in range(15)]
    for i in range(15):
        map[0][i] = 1
        map[14][i] = 1
        map[i][0] = 1
        map[i][14] = 1

    checkWallCurrentNodeUp()
    checkWallCurrentNodeRight()
    map[5][9] = 0
```

Initially, the maze needs to be created so that when starting, the starting position is set, and the orientation is also set then. It checks nearby and starts its algorithm searching and mapping.

Explanation of final algorithm and searching method and conclusion:

My final program is significantly longer than most other competitors in the competition; I designed my AI to be entirely self-correcting and able to function within any maze provided it needed to search for an extended period of time; with its self-correcting functionality, it can go long periods of searching without correction. During testing, I got it to run for 15 minutes cycling the maze just to test its correctional abilities.

The algorithms used are differing because I wanted to select and use a uniquely tuned algorithm for each stage of the maze competition; the first is a weighted flood search with support from a breadth-first search if it gets stuck, and to solve the maze I used another breadth-first search algorithm to find the most efficient path to move the robot to victory.

In conclusion, I learned quite a bit from this example, and I also think that I could make a reasonably competent AI if it needed to do something simple. I do think that because of the size of the maze, we were not forced to use any one algorithm because the maze is so tiny the change in efficiency is minute. It does not affect the time spent on the maze unless we choose an algorithm that would provide anything less than the optimal path. I do think that this competition could be significantly improved by either providing a variety of robots, providing more time to program our AI aswell, or potentially providing assistance with our testing and examples.

If I could redo this project and enforce the same idea of algorithms in a different format but still allow competition. I do think making something simple like a cat and mouse AI would also be very fun, or making an AI for some other made-up simple game that involves pathing away from a set of catchers that need to map the maze. In contrast, the mouse runs away, and as for the assignment, you would need to program both a catcher and a mouse.

Bibliography Following IEEE Standard:

[1] N. Kumar, “A Review of Various Maze Solving Algorithms Based on Graph Theory,” *IJSRD*, vol. 6, no. 12, Mar. 2019, [Online]. Available: https://www.researchgate.net/publication/331481380_A_Review_of_Various_Maze_Solving_Algorithms_Based_on_Graph_Theory

[2] S. Ayrinhac, “Electric current solves mazes,” *Phys. Educ.*, vol. 49, no. 443, doi: 10.1088/0031-9120/49/4/443.

[3] F. Meyer, “Flooding edge or node weighted graphs,” pp. 1–165, Mar. 2013, doi: https://doi.org/10.48550/arXiv.1305.5756.

[4] Community, “Maze Solving Algorithm.” [Online]. Available: https://encyclopedia.pub/entry/33079

[5] “MicroMouse Competition Rules IEEE R2 SAC 2020.” IEEE, 2020. [Online]. Available: https://attend.ieee.org/r2sac-2020/wp-content/uploads/sites/175/2020/01/MicroMouse_Rules_2020.pdf

[6] V. R, “Big o Cheatsheet - Data structures and Algorithms with thier complexities,” *hackerearth*. https://www.hackerearth.com/practice/notes/big-o-cheatsheet-series-data-structures-and-algorithms-with-thier-complexities-1/