Naoki Shibuya

March 20, 2016

Capstone Project

Machine Learning Engineer Nanodegree

Plot and Navigate a Virtual Maze

Definition

Project Overview

A robot mouse in a virtual maze finding an optimal way to the destination by its own – that's what I programed in this project which took an inspiration from the micro mouse competition.

Problem Statement

The rule is simple: in the first run, the robot mouse tries to map out the maze to not only find the center, but also figure out the best paths to the center. The robot must enter to the goal within the time limit but it is free to continue exploring the maze after finding the goal. In subsequent runs, the robot mouse is brought back to the start location. It must attempt to reach the center in the fastest time possible, using what it has previously learned.

A simplified model of the world is provided along with specifications for the maze and robot. My main objective is to implement a logic that achieves the fastest times possible in a series of test mazes.

Scoring

The robot's score for the maze is equal to the number of time steps required to execute the second run, plus one thirtieth the number of time steps required to execute the first run. A maximum of one thousand time steps is allotted to complete both runs for a single maze.

Analysis

Data Exploration and Visualization

The shape of every test mazes is a square. The start location is always at the left bottom corner (rows-1, 0), and the goal room always occupies 4 cells in the center.

The robot has three obstacle sensors, mounted on the front of the robot, its right side, and its left side. Obstacle sensors detect the number of open squares in the direction of the sensor. It is assumed that the robot's turning and movement is perfect.

At the start location, both left and right sides have walls so that it can only move forward. The sensors will return the distance between the robot and walls in a tuple as in (left distance, forward distance, right distance).

On each time step of the simulation, the robot may choose to rotate clockwise or counterclockwise ninety degrees, then move forwards or backwards a distance of up to three units.

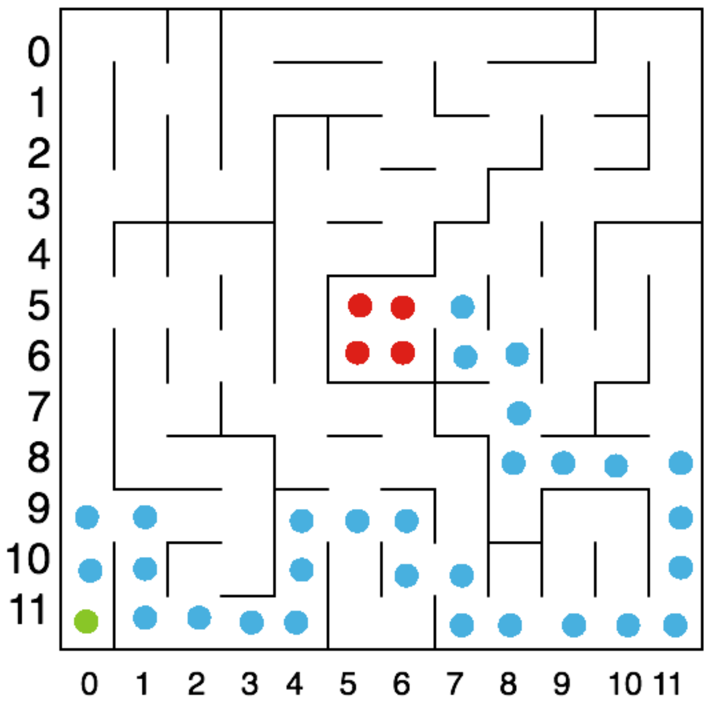
If the robot tries to move into a wall, the robot stays where it is. After movement, one time-step has passed, and the sensors return readings for the open squares in the robot's new location and/or orientation to start the next time unit.

Rotation is expected to be an integer taking one of three values: -90, 90, or 0, indicating a counterclockwise, clockwise, or no rotation, respectively. Movement follows rotation, and is expected to be an integer in the range [-3, 3] inclusive. The robot will attempt to move that many squares forward (positive) or backwards (negative), stopping movement if it encounters a wall.

Maze 01 (12x12)

The start location is (11, 0) and it is shown with the green ball. The goal area is located in the center and it is shown with the red balls. At the start location, the robot is facing north and its sensors will return (0, 11, 0).

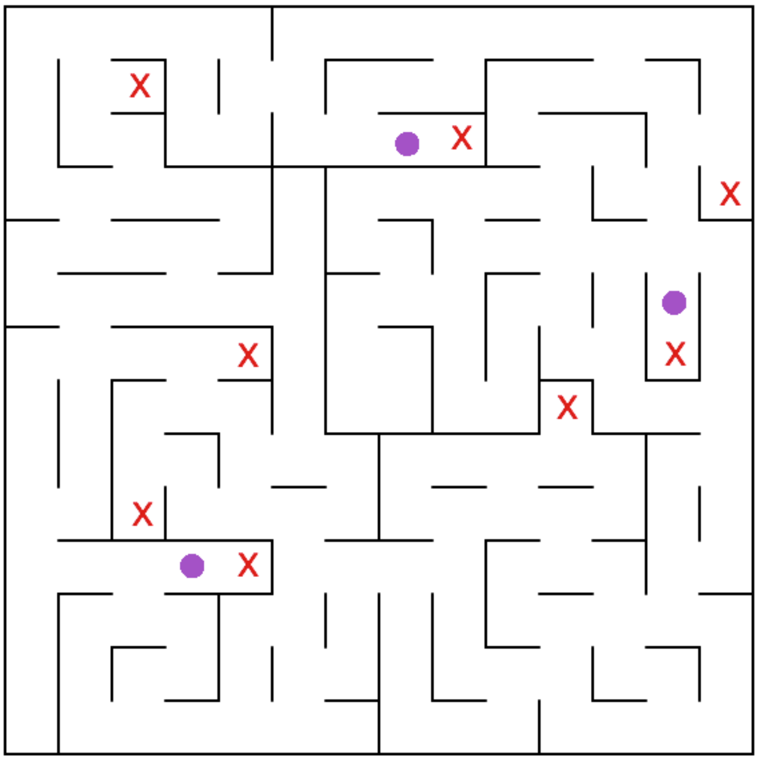
The shortest path from the start location to the goal area is indicated the blue balls. It takes 30 single steps from the start location to the goal area. The robot is however allowed to take maximum 3 steps in one time step. Therefore, the minimum time steps required to arrive the goal should be 17. This also means the robot should prefer straight path than left/right turns.



Maze 02 (14x14)

The robot should identify and avoid dead ends. The robot should recognize any one-way path to dead ends as well.

In this maze visualization, I marked dead ends with red X marks and one way path to the dead ends with purple balls.

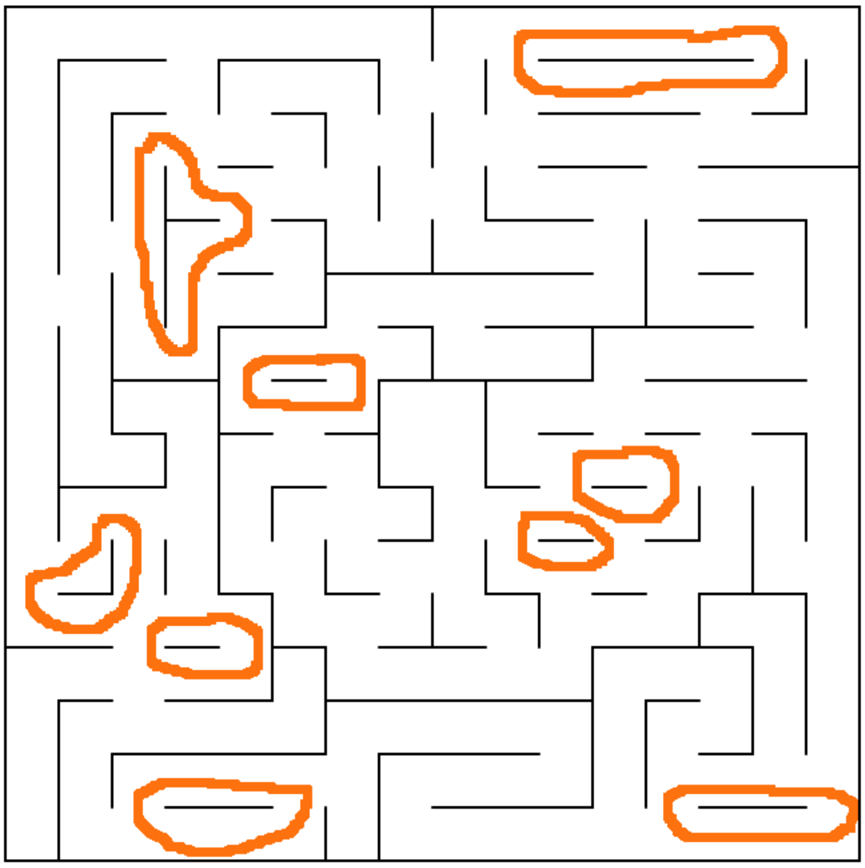


Maze 03 (16x16)

The robot should avoid going through a loop many times.

In this maze, I indicated some of the potential loops in orange lines. The robot may fall into the same path again and again there.

During the first run, the robot should keep track of how often it has visited each location in the maze and prefers to explore where it visited less often.



Algorithms and Techniques

If the robot explores the entire maze, we can use A* algorithm to find the shortest path from the start location to the goal area. Therefore, in the first run, the robot should try exploring the maze as much as possible and expand the mapping area so that, in the second run, the robot can apply the A* search algorithm to find the optimal path and moves. On the other hand, the robot should try reaching the goal as fast as possible in the first run as it affects the score value and if too much time is spent in the first run, it may not be able to leave enough time for the second run.

For the first run, I will use the following techniques for the robot controller to explore the maze and expand the mapping area while seeking the goal area:

- Random move
- Random move with dead-end path detection
- Counting number of visits for each location in order to expand into less visited area
- The counting logic with heuristic prediction of the distance to the goal

The random move controller will take the robot to different paths randomly. It is not most efficient way to expand the mapping area but it gives the baseline performance to compare with more advanced techniques. Also, this controller will prove that the robot movements and rotations are handling walls properly.

The dead-end path detection will prevent the robot to enter dead-ends more than once making it more efficient to expand the mapping area than the pure random controller.

The counting number of visits for each location will give the robot chances to move to less frequently visited locations. It will also make the robot moves out of loops.

The heuristic values will be used to make the robot move towards the goal area making it faster to reach the goal area.

For the second run, I will use A* search on the mapped area from the first run in order to find the optimal path/moves to the goal.

In A* search, G values give the cost of reaching to each location in the maze from the start position. The turning left or right is given higher cost than the forward move. Heuristic values gives the distance of each location in the maze from the goal area. A* search uses the F values which are combination of G values and Heuristic values to find the optimal path from the start location to the goal area.

As the maze structure is well defined, I will use the test maze data to test the A* search program. I'll provide a separate python script to run the test without using the robot tester program.

Benchmark

I'll provide different controllers (Random, Dead-End, Counter, Heuristic) to compare the scores which should improve as more advanced techniques are introduced. The random controller will set the worst score benchmark. Later, I'll discuss the A* search results with the full maze details, which will set the optimal path/moves as the best benchmark.

Methodology

Data Pre-processing

The maze specification and robot's sensor data is provided and 100% accurate. Therefore, there is no data pre-processing is required.

Implementation

Utilities

In utility.py, I have implemented utilities to handle common tasks.

Delta is an array of directions. Each item in Delta is a pair of (row direction, column direction). For example, [-1, 0] means North direction.

```
Delta = [[-1,  0], # go north
         [ 0,  1], # go east
         [ 1,  0], # go south
         [ 0, -1]] # go west
```

A change in direction can be expressed by an index move in Delta. For example, a right turn means adding 1 to the index. When the current direction is North (index=0), a right turn results in the direction change to East (index=1).

Steering is an Enum class for direction change. Rather than remembering what -1 means, I can simply use Steering.L to mean a left turn.

```
class Steering(Enum):
    L, F, R = (-1,0,1) # Left, Forward, Right

    def __str__(self):
        return self.name
```

Direction is an Enum class for direction. The enum value of this class corresponds to the index values in Delta so that I do not need to remember the actual index value and use the Enum name instead. This class handles direction manipulation operations such as reversing direction or adjusting to change the direction by a Steering value.

```
class Direction(Enum):
    N, E, S, W = range(4) # North, East, South, West

    def reverse(self):
        return Direction((self.value+2)%4)

    def adjust(self, steering):
        return Direction((self.value+steering.value)%4)
    def delta(self):
        return Delta[self.value]

    def steer(self, direction):
        diff = direction.value - self.value
        if diff ==3:
            diff = -1
        if diff ==-3:
            diff = 1
        return Steering(diff)

    def __str__(self):
        return self.name
```

The robot decides direction and moves its location. **Heading** is a class that encapsulates both direction and location. It is mutable and as such all methods will return a new Heading object.

```
class Heading(object):
    def __init__(self, direction, location):
        self.direction = direction
        self.location = location

    def __str__(self):
        return '{} @ ({:>2d},{:>2d})'.format(
            self.direction.name, self.location[0], self.location[1])

    def adjust(self, steering, movement):
        direction = self.direction.adjust(steering)
        delta = direction.delta()
        location = [ self.location[i]+delta[i]*movement for i in range(2) ]
        return Heading(direction, location)

    # move forward
    def forward(self, movement=1):
        return self.adjust(Steering.F, movement)

    # move to left
    def left(self, movement=1):
        return self.adjust(Steering.L, movement)

    # move to right
    def right(self, movement=1):
        return self.adjust(Steering.R, movement)

    # move backward with turning
    def backward(self, movement=1):
        return self.reverse().forward(movement)

    # only reverse the direction
    def reverse(self):
        return Heading(self.direction.reverse(), self.location)
```

The robot's sensor values are given in a tuple format. Again, directly handling it can be error-prone. Making the code more human readable is a way to avoid such problems.

Sensor is a class that encapsulates sensor values. I can get a distance to a wall by specifying the steering value, and also find out if the robot is in a dead-end or not.

```
class Sensor:
    def __init__(self, sensors):
        self.sensors = sensors

    def distance(self, steering):
        steering_sensor_index_map = {
            Steering.L : 0,
            Steering.F : 1,
            Steering.R : 2
        }
        return self.sensors[steering_sensor_index_map[steering]]

    def isDeadEnd(self):
        return max(self.sensors)==0

    # both sides are walls
    def isOneWay(self):
        return self.sensors[0]==0 and self.sensors[1]>0 and self.sensors[2]==0

    def __str__(self):
        return str(self.sensors)
```

Goal is a class that encapsulates the goal area in a maze so that I can check whether a location is in the goal area or not.

```
class Goal(object):
    def __init__(self, rows, cols):
        self.goal_row_max = rows/2
        self.goal_row_min = rows/2-1
        self.goal_col_max = cols/2
        self.goal_col_min = cols/2-1

    def isGoal(self, location):
        row, col = location
        return self.goal_row_min <= row and row <= self.goal_row_max and \
               self.goal_col_min <= col and col <= self.goal_col_max
```

Grid encapsulates two dimensional array so that I can associate maze locations to calculated values. I can get and set values using getValue and setValue methods, making the code readable that using the array index syntax.

```
class Grid(object):
    def __init__(self, rows, cols ,init_val):
        self.rows = rows
        self.cols = cols
        self.grid = [ [ copy.deepcopy(init_val) for c in range(cols) ] for r
in range(rows) ]
        self.shape = (rows, cols)

    def __getitem__(self, row):
        return self.grid[row]

    def getValue(self, location):
        return self.grid[location[0]][location[1]]

    def setValue(self, location, value):
        self.grid[location[0]][location[1]] = value

    def isValid(self, location):
        row, col = location
        return 0 <= row and row < self.rows and 0 <= col and col < self.cols
```

There are several subclass of Grid. Namely,

- **Mapper** for mapping the maze,
- **Counter** for counting the number of visits for each cell,
- **DeadEnd** for marking dead-end paths, and
- **Heuristic** to generate and holds the heuristic values

A* search

In planner.py, I've implemented A* star search (findOptimalMoves) to calculates the optimal moves from the start location to the goal area. Having a standalone program makes it easy to test the search algorithm directly with the test maze files.

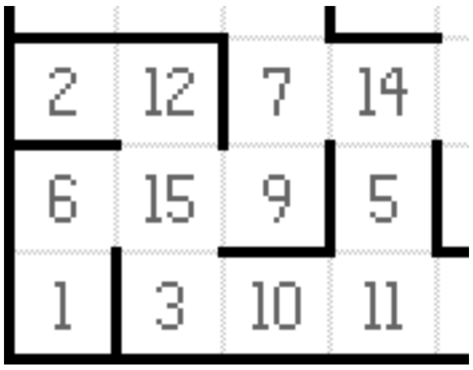
The program can be run by the planner.sh script as follows:

```
./plan.sh <maze_number>
```

To find the optimal path and moves for the test_maze_01.txt, use the following:

```
./plan.sh 01
```

Mapper class can read the test maze file format so that the search algorithm knows where it can move to.



Note: it reads the maze file only when testing the A* search logic. It will not do so while running the tester program to test the actual robot controllers.

A* star search then uses Heuristic values and g-values to calculate f-values. The optimal path will be printed into the console as shown below:

```
-- Path --
 , , , , , , , , , , , 
 , , , , , , , , , , , 
 , , , , , , , , , , , 
 , , , , , , , , , , , 
 , , , , , , , , , , , 
 , , , , , ,*,W, , , , 
 , , , , , , ,N,W, , , 
 , , , , , , , ,N, , , 
 , , , , , , , ,N,W,W,W
E,S, , ,E,E,S, , , , ,N
N,S, , ,N, ,E,S, , , ,N
N,E,E,E,N, , ,E,E,E,E,N
Path Length: 30
```

The optimal moves are returned in a list of (steering, movement) pairs. The list of moves can be applied from the start location to reach the goal with the minimum steps.

A sample is shown below for the test maze 01. The optimal path length is 30 but the actual moves required is only 17 since the robot can take up to 3 steps in one direction.

```
-- Moves --
(F,2)
(R,1)
(R,2)
(L,3)
(L,2)
(R,2)
(R,1)
(L,1)
(R,1)
(L,3)
(F,1)
(L,3)
(L,3)
(R,2)
(L,1)
(R,1)
(L,1)
# of Moves: 17
```

Refinement

Robot Controllers

In controller.py, I defined a Controller base class which declares methods that the robot class interact with. There are various sub-classes of this class for specific logic implementations.

The following is a list of controllers used for the first run to explore the maze:

- **Controller_Exploration** a base class for first run controllers
- **Controller_Random** extends Controller_Exploration to implement the random move
- **Controller_DeadEnd** extends Controller_Random with dead-end path detection
- **Controller_Counter** extends Controller_DeadEnd with the location visit counter allowing it to move to less visited locations for better exploration/expansion and avoid loops
- **Controller_Heuristic** extends Controller_Counter using the heuristic to decide direction when two possible directions has same counter value to reach the goal faster

In the second run, the robot internally switches to the following controller to reach to the goal following optimal moves calculated with the mapped area of the maze.

- **Controller_Exploitation** uses the findOptimalMoves to get a list of optimal moves to follow based on the available information on the maze structure from the first run

Robot

The tester program can be run as follows:

```
./run.sh <controller_name> <maze_number> (<tick_interval_in_seconds>)
```

For example, to run the random controller with maze 02:

```
./run.sh random 02
```

For debugging purpose, you can specify a delay seconds between each time step. The following adds 1 second between each time step so that you can actually follows the log messages.

```
./run.sh random 02 1
```

The following is a list of controller names:

- random
- deadend
- counter
- heuristic

Results

Model Evaluation and Validation

Optimal Moves

The optimal moves are measured by using the A* search.

Test maze	Path Length	Required Moves
01	30	17
02	43	23
03	49	25

The above numbers can be achieved if the robot has 100% mapping coverage of the maze as all robots are using the same search implementation in the 2nd run.

<u>Maze 01 Optimal Moves</u>

```
-- Path --
 , , , , , , , , , , ,
 , , , , , , , , , , ,
 , , , , , , , , , , ,
 , , , , , , , , , , ,
 , , , , , , , , , , ,
 , , , , , ,*,W, , , ,
 , , , , , , ,N,W, , ,
 , , , , , , , ,N, , ,
 , , , , , , , ,N,W,W,W
E,S, , ,E,E,S, , , , ,N
N,S, , ,N, ,E,S, , , ,N
N,E,E,E,N, , ,E,E,E,E,N

Path Length! 30

-- Moves --
(F,2)
(R,1)
(R,2)
(L,3)
(L,2)
(R,2)
(R,1)
(L,1)
(R,1)
(L,3)
(F,1)
(L,3)
(L,3)
(R,2)
(L,1)
(R,1)
(L,1)

# of Moves! 17
```

There are alternate paths that have the same length (=30). The above move is selected as the A* search gives preference to straight paths by adding extra move cost to left and right turns.

For example, the following path's length is 30 but the number of moves is 20 which is worse than the optimal moves by 3 steps.

<u>Maze 01 Non-Optimal Moves</u>

```
-- Path --
 , , , , , , , , , , , 
 , , , , , , , ,E,S, , 
 , , , , ,E,E,E,N,S, , 
 , , , ,E,N, , ,S,W, , 
 , , , ,N, , ,S,W, , , 
E,E,S, ,N, ,*,W, , , , 
N, ,E,S,N, , , , , , , 
N, , ,E,N, , , , , , , 
N, , , , , , , , , , , 
N, , , , , , , , , , , 
N, , , , , , , , , , , 
N, , , , , , , , , , , 

Path Length! 30

-- Moves --
(F,3)
(F,3)
(R,2)
(R,1)
(L,1)
(R,1)
(L,1)
(L,3)
(F,1)
(R,1)
(L,1)
(R,3)
(L,1)
(R,1)
(R,2)
(R,1)
(L,1)
(R,1)
(L,1)
(R,1)

# of Moves! 20
```

<u>Maze 02 Optimal Moves</u>

```
-- Path --
 , , , , , , , , , , , , , 
 , , , , , , , , , , , , , 
 , , , , , , , , , , , , , 
 , , , , , , , , , , , , , 
 , , , , , , , ,S,W,W,W,W,W
 , , , , , ,S,W,W, , , , ,N
E,E,E,S, , ,*, , , , , , ,N
N, , ,E,S, , , , , , , , ,N
N, , , ,S, , , , , , , ,E,N
N, , , ,E,S, , , , , , ,N, 
N, , , , ,E,E,E,S, , , ,N, 
N, , , , , , , ,S, ,E,E,N, 
N, , , , , , , ,E,E,N, , , 
N, , , , , , , , , , , , , 

Path Length! 43

-- Moves --
(F,3)
(F,3)
(F,1)
(R,3)
(R,1)
(L,1)
(R,2)
(L,1)
(R,1)
(L,3)
(R,2)
```

```
(L,2)
(L,1)
(R,2)
(L,3)
(R,1)
(L,3)
(F,1)
(L,3)
(F,2)
(L,1)
(R,2)
(L,1)

# of Moves! 23
```

<u>Maze 03 Optimal Moves</u>

```
-- Path --
E,E,E,E,E,E,E,S, , , , , , , , ,
N, , , , , , ,E,S, , , , , , , ,
N, , , , , , , ,E,E,E,E,S, , , ,
N, , , , , , , , , , , ,S, , , ,
N, , , , , , , , , , , ,S, , , ,
N, , , , , , , , , , , ,E,E,S, ,
N, , , , , , , , , , ,S,W,W,W, ,
N, , , , , , , , , , ,S, , , , ,
N, , , , , , , ,*, ,S,W, , , , ,
N, , , , , , , ,N,W,W, , , , , ,
N, , , , , , , , , , , , , , , ,
N,W,W, , , , , , , , , , , , , ,
E,E,N, , , , , , , , , , , , , ,
N, , , , , , , , , , , , , , , ,
N, , , , , , , , , , , , , , , ,
N, , , , , , , , , , , , , , , ,

Path Length! 49

-- Moves --
(F,3)
(R,2)
(L,1)
(L,2)
(R,3)
(F,3)
(F,3)
(F,2)
(R,3)
(F,3)
(F,1)
(R,1)
(L,1)
(R,1)
(L,3)
(F,1)
(R,3)
(L,2)
(R,1)
(R,3)
(L,2)
(R,1)
(L,1)
(R,2)
(R,1)

# of Moves! 25
```

Again, there can be a different path with the same number of moves. For example, the following path length is 45 (shorter than above) and it takes 25 moves (the same). The reason this was not chosen is because the list of moves contains less number of forward moves. In other words, there are more left and right turns which costs more than forward moves (in the physical micro mouse competition, it’d take more time to make turns than moving straight).

```
-- Path --
 , , , , , , , , , , , , ,
 , , , , , , , , , , , , ,
 , , , , , , , , , , , , ,
 , , , , , , , ,S,W,W, , ,
 , , , , , , , ,S, ,N, , ,
 , , , , , ,S,W,W, ,N, , ,
 , , , , , ,*, , , ,N,W, ,
 , , , , , , , , , , ,N,W,W
 , , , , , , , , , , , ,E,N
 , , , , , , , , , , , ,N,
E,E,S, , ,E,E,E,S, , , ,N,
N, ,E,S, ,N, , ,S, ,E,E,N,
N, ,S,W, ,N, , ,E,E,N, , ,
N, ,E,E,E,N, , , , , , , ,

Path Length! 45

-- Moves --
(F,3)
(R,2)
(R,1)
(L,1)
(R,1)
(R,1)
(L,1)
(L,3)
(L,3)
(R,3)
(R,2)
(L,2)
(L,1)
(R,2)
(L,3)
(R,1)
(L,1)
(L,2)
(R,1)
(L,1)
(R,3)
(L,2)
(L,2)
(R,2)
(L,1)

# of Moves! 25
```

Random and Dead-End controllers

Both the random and the dead-end controllers use random moves chosen from the available directions. As such, they give different outcome for every run. Moreover, it may or may not reach to the goal. I took 10 trial results to evaluate the average numbers for them.

Random Controller

The random controller does reach to the goal but not always. As the maze size increases, the success rate goes down. The robot turns left or right randomly at the dead-ends but it does not recognize the one-way path to them. Hence, it may repeatedly visit the same dead-ends over and over. After examining the log details, it became evident that the random logic spent lots of time in the dead end paths. Note: the below average values exclude rows with Goal=No.

Test Maze 01 Results

Trial	1st Run			2nd Run			Score
	Goal?	Moves	Coverage	Goal?	Path Length	Moves	
1	Yes	831	95.14%	Yes	30	17	44.767
2	Yes	210	66.67%	Yes	34	21	28.067
3	Yes	111	38.19%	Yes	34	21	24.767
4	Yes	108	43.06%	Yes	30	21	24.667
5	Yes	179	56.25%	Yes	30	17	23.033
6	Yes	638	89.58%	Yes	30	17	38.333
7	Yes	60	36.11%	Yes	32	19	21.067
8	No	n/a	n/a	n/a	n/a	n/a	n/a
9	Yes	256	72.92%%	Yes	36	22	30.600
10	No	n/a	n/a	n/a	n/a	n/a	n/a
Average	80.00%	299.125	60.71%	80.00%	32	19.375	29.413

Test Maze 02 Results

Trial	1st Run			2nd Run			Score
	Goal?	Moves	Coverage	Goal?	Path Length	Moves	
1	Yes	399	86.22%	Yes	43	26	39.367
2	Yes	172	52.55%	Yes	49	31	36.800
3	Yes	258	55.61%	Yes	47	31	39.667
4	Yes	251	50.51%	Yes	47	30	38.433
5	No	n/a	n/a	n/a	n/a	n/a	n/a
6	No	n/a	n/a	n/a	n/a	n/a	n/a
7	Yes	737	72.45%	Yes	43	25	49.633
8	Yes	805	89.29%	Yes	43	23	49.900
9	No	n/a	n/a	n/a	n/a	n/a	n/a
10	No	n/a	n/a	n/a	n/a	n/a	n/a
Average	60.00%	437	67.77%	60.00%	45.33	27.67	42.300

Test Maze 03 Results

Trial	1st Run			2nd Run			Score
	Goal?	Moves	Coverage	Goal?	Path Length	Moves	
1	No	n/a	n/a	n/a	n/a	n/a	n/a
2	No	n/a	n/a	n/a	n/a	n/a	n/a
3	No	n/a	n/a	n/a	n/a	n/a	n/a
4	Yes	618	84.38%	Yes	51	26	46.667
5	No	n/a	n/a	n/a	n/a	n/a	n/a
6	No	n/a	n/a	n/a	n/a	n/a	n/a
7	No	n/a	n/a	n/a	n/a	n/a	n/a
8	Yes	226	53.12%	Yes	63	36	43.600
9	Yes	361	66.41%	Yes	49	26	38.100
10	Yes	519	75.78%%	Yes	51	31	48.367
Average	40.00%	431	67.97%	40.00%	53.5	29.75	44.184

Dead-End Controller

The dead-end controller avoids dead-end paths by keep track of dead-end paths. As a result, the success rate is higher than the random controller. Having said that, it can still fail to reach the goal from time to time due to the random move choice. The mapping coverage is slightly higher than the random controller, too. However, higher coverage does not always translates to better scoring in the 2nd run, indicating the robot needs better logic than just randomly wandering about.

It was observed that the dead-end controller spent lots of time in loops. We need a controller that avoid going through the same path again and again.

Test Maze 01 Results

Trial	1st Run			2nd Run			Score
	Goal?	Moves	Coverage	Goal?	Path Length	Moves	
1	Yes	62	38.89%	Yes	50	32	34.133
2	Yes	282	81.25%	Yes	36	23	32.467
3	Yes	426	81.94%	Yes	30	17	31.267
4	Yes	227	81.25%	Yes	36	23	30.633
5	Yes	545	92.36%	Yes	30	17	35.233
6	Yes	267	66.67%	Yes	30	17	25.967
7	Yes	147	56.25%	Yes	38	22	26.967
8	Yes	137	60.42%	Yes	38	25	29.633
9	Yes	972	97.92%	Yes	30	17	49.467
10	Yes	258	76.39%	Yes	32	21	29.667
Average	100.00%	332.300	73.33%	100.00%	35.00	21.400	32.543

Test Maze 02 Results

Trial	1st Run			2nd Run			Score
	Goal?	Moves	Coverage	Goal?	Path Length	Moves	
1	Yes	440	67.86%	Yes	45	25	39.733
2	Yes	180	50.51%	Yes	49	31	37.067
3	Yes	533	79.59%	Yes	43	25	42.833
4	Yes	827	91.33%	Yes	47	28	55.633
5	No	n/a	n/a	n/a	n/a	n/a	n/a
6	Yes	97	41.33%	Yes	61	36	39.300
7	Yes	517	82.14%	Yes	43	27	44.300
8	No	n/a	n/a	n/a	n/a	n/a	n/a
9	Yes	490	59.18%	Yes	43	28	44.400
10	No	n/a	n/a	n/a	n/a	n/a	n/a
Average	70.00%	440.571	67.42%	70.00%	47.29	28.571	43.324

Test Maze 03 Results

Trial	1st Run			2nd Run			Score
	Goal?	Moves	Coverage	Goal?	Path Length	Moves	
1	Yes	348	69.53%	Yes	59	31	42.667
2	Yes	178	48.83%	Yes	57	33	39.000
3	Yes	188	39.45%	Yes	49	28	34.333
4	Yes	913	85.55%	Yes	51	25	55.500
5	Yes	333	59.77%	Yes	53	29	40.167
6	Yes	584	85.16%	Yes	51	28	47.533
7	Yes	703	85.55%	Yes	51	26	49.500
8	Yes	473	76.56%	Yes	51	29	44.833
9	Yes	305	58.59%	Yes	59	31	41.233
10	Yes	676	87.89%	Yes	53	29	51.600
Average	100.00%	470.100	69.69%	100.00%	53.40	28.900	44.637

Dead-End Controller Part 2

I've experimented with the dead-end controller by making it back off from the dead-ends with negative movement value. The idea is using one back off move rather than turn left or right twice to move away from the dead-ends. This proves to be a bad idea after observing the log. When the robot backs off and moves out of a dead-end, it can only move either left or right as the forward move will bring it back into the dead-end path, limiting the move options to maximum 2. It is actually better to turn left or right at dead-ends just like the original controller does. Then, when the robot is moving out of the dead-end path, it has maximum 3 directional options making the exploration more effective.

Counter and Heuristic controllers

The counter controller and the heuristic controller have no randomness. They give the same results every time. Therefore, I measured their performances by one trial for each controller.

Counter Controller

The counter controller keeps track of how often each location has been visited and explores less visited locations. The coverage rate is high with less moves than previous controllers in the maze 01 and the maze 02. The controller is avoiding loops very well.

However, it does a bad job in the test maze 03 because it is not aware of where the goal is and earnestly exploring the maze away from the goal.

<u>Test Maze 01 Results</u>

Trial	**1st Run**			**2nd Run**			**Score**
	Goal?	**Moves**	**Coverage**	**Goal?**	**Path Length**	**Moves**	
1	Yes	170	86.11%	Yes	32	17	22.733

<u>Test Maze 02 Results</u>

Trial	**1st Run**			**2nd Run**			**Score**
	Goal?	**Moves**	**Coverage**	**Goal?**	**Path Length**	**Moves**	
1	Yes	378	91.84%	Yes	45	27	39.667

<u>Test Maze 03 Results</u>

Trial	**1st Run**			**2nd Run**			**Score**
	Goal?	**Moves**	**Coverage**	**Goal?**	**Path Length**	**Moves**	
1	Yes	806	98.83%	Yes	51	25	51.933

Heuristic Controller

The heuristic controller uses the heuristic values to guide the robot to move towards the goal. When there are two directions where both has the same counter value, it will choose the one with a smaller heuristic value.

The result is much better than any other controllers in all mazes.

<u>Test Maze 01 Results</u>

Trial	**1st Run**			**2nd Run**			**Score**
	Goal?	**Moves**	**Coverage**	**Goal?**	**Path Length**	**Moves**	
1	Yes	103	54.17%	Yes	34	18	21.500

Test Maze 02 Results

Trial	1st Run			2nd Run			Score
	Goal?	Moves	Coverage	Goal?	Path Length	Moves	
1	Yes	183	68.37%	Yes	47	29	35.167

Test Maze 03 Results

Trial	1st Run			2nd Run			Score
	Goal?	Moves	Coverage	Goal?	Path Length	Moves	
1	Yes	109	35.55%	Yes	59	33	36.700

Justification

The best performing controller is the heuristic controller in terms of the score value. The below table compares the result with the best path length and optimal moves for each test maze.

Test maze	Best Path Length	Best Path Optimal Moves	Heuristic Controller Path Length	Heuristic Controller Moves
01	30	17	34	18
02	43	23	47	29
03	49	25	59	33

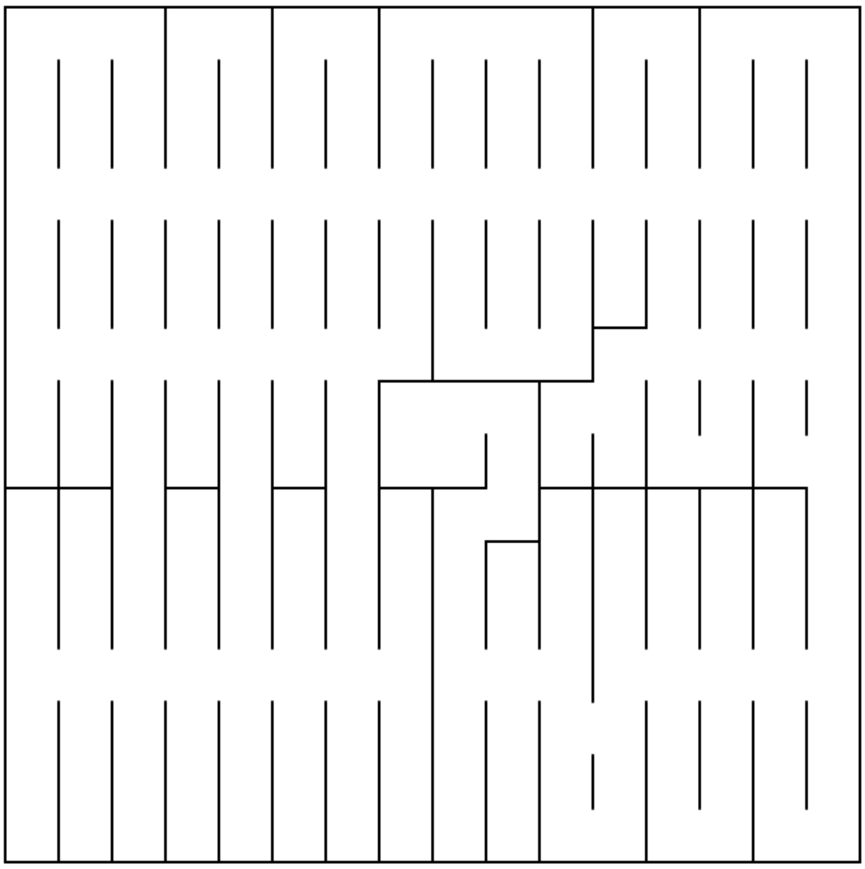
The heuristic controller uses much less moves in the first run than any other controllers and yet it can achieve good moves to take the robot to the goal in the second run.

Conclusion

Free-Form Visualization

It is easily possible to make a maze more difficult for particular expansion/goal-seeking logic. I came up with a new maze (Maze 04) to prove that point as shown below.

Maze 04 (16x16)



This maze has many dead-ends and loops. The random controller performs very badly in this maze with low success rate. The dead-end controller performs much better but not as good as the counter controller since there are many loops in this maze. The heuristic controller performs much better than the random controller and the dead-end controller.

<u>Test Maze 04 Results for Heuristic Controller</u>

Trial	1st Run			2nd Run			Score
	Goal?	Moves	Coverage	Goal?	Path Length	Moves	
1	Yes	370	76.17%	Yes	52	25	37.400

Unlike with the test maze 03 (109 moves in the first run), the heuristic controller spent 370 moves in the first run, indicating the heuristic controller was not very fast to find the goal in this maze.

This result is expected as I actually made the maze really hard for the heuristic controller. The below diagram of the optimal moves for the test maze 04 shows the reason why.

```
-- Path --
 , , , , , , , , , , , , , , , 
 , , , , , , , , , , , , , , , 
 , , , , , , , , , , , , , , , 
 , , , , , ,E,E,E,E,E,E,E,E,E,S
 , , , , , ,N, , , , , , , , ,S
 , , , , , ,N, , , , , , , , ,S
 , , , , , ,N, , , , , , , , ,S
 , , , , , ,N, ,*,W, , , , , ,S
 , , , , , ,N, , ,N, , , , , ,S
 , , , , , ,N, ,E,N, , , , , ,S
 , , , , , ,N, ,N, , , , , , ,S
 , , , , , ,N, ,N, , , , , , ,S
E,E,E,E,E,E,N, ,N,W,W,S,W,W,W,W
N, , , , , , , , , ,N,W, , , , 
N, , , , , , , , , , , , , , , 
N, , , , , , , , , , , , , , , 

Path Length! 52

-- Moves --
(F,3)
(R,3)
(F,3)
(L,3)
(F,3)
(F,3)
(R,3)
(F,3)
(F,3)
(R,3)
(F,3)
(F,3)
(R,3)
(F,1)
(L,1)
(R,1)
(R,1)
(L,2)
(R,3)
(R,1)
(L,2)
(L,1)

# of Moves! 22
```

The optimal path to the goal requires the robot to go almost full circle around the goal area, making the heuristic values less useful.

The counter controller, however, performs better than the heuristic controller in this maze:

Test Maze 04 Results for Counter Controller

Trial	1st Run			2nd Run			Score
	Goal?	Moves	Coverage	Goal?	Path Length	Moves	
1	Yes	145	48.83%	Yes	74	31	35.900

The counter controller spent only 145 moves in the first run. The second run of the counter controller is much worse than that of the heuristic controller. But the counter controller outperforms the heuristic controller in the first run by large.

The reason for the counter controller to performs better than the heuristic controller in this maze was because it has no bias for the goal location which is advantage of the heuristic controller in other mazes.

Reflection

The initial challenge for me was to divide the project into smaller problems to tackle with. I've decided to have separate test program for the A* search program as I realized it does not require a running robot to test the search algorithm. Then, I've divided each enhancements to the robot controller into different python classes making it easier to add improvements in terms of coding and also the actual test scores. Throughout the project, I was making common tasks into utility classes so that I do not need to write similar logic or handling in different places (i.e. Direction, Steering, Sensor, Grid).

The next challenge for me was to find problems in expanding the maze mapping area. I had issues like dead-ends and looping. Over the time, I've added effective logging to analyse the robot behaviour and look for potential issues. This was largely a trial-and-error process for me since I did not have much experience in the maze solving problem before. It was a great learning process for me.

Finally, the most difficult and interesting part was how to expand the mapping area of the maze while reaching to the goal at minimum time required in the first run. Unlike in the second run, where the robot simply uses A* search to find the optimal path and travel to the goal accordingly, the first run was full of challenges due to the unknown area to be explored. Overall, I built a series of controllers to add improvement bit by bit making the process simpler than without such structural approach to solve the main problem. I believe the final controller gives fine performance in the real micro mouse scenario with some improvement for continuous domain support.

Improvement

In this project, everything (time, location, move and turn) is in a discrete domain. In the real micro mouse competition, everything is in a continuous domain.

For example, the distance from the robot to the wall is measured in continuous value with some sensor errors. The robot movement itself would have some randomness. Therefore, the robot would need to perform SLAM (simultaneous localization and mapping) to explore the maze. Moreover, the robot needs to use PID control to continuously adjust the direction and turns so that it can wander around in the maze without colliding with the walls. The speed needs to be controlled rather than just number of steps. Turns will be continuous rotations. Moreover, the robot may be able to move diagonally rather than zigzag which is not allowed in the discrete domain.

Talking about the real micro mouse competition, the fact that the robots are physical adds many more complexity. The path finding logic is probably one of the easiest part of the whole robot construction. There are many aspects to take care in physical robots: what sensors to use, what kind of motors and how heavy it can be, how much memory size available to use, etc. Maybe I could have a sensor rotating on top of the robot mapping neighboring areas simultaneously just like a google car. The possibilities are endless.