|  |  |
| --- | --- |
| **Capstone Project** | Muhammad Zeeshan Karamat |
| **Machine Learning Engineer Nanodegree** | July 13, 16 |

Plot and Navigate a Virtual Maze

# Definition

**Project Overview:**

A robotic mouse in an unknown virtual maze, is programmed to firstly plot the unknown environment and then find the optimal path from start location to the center of the virtual maze. [This video](https://www.google.com/url?q=https://www.youtube.com/watch?v%3D0JCsRpcrk3s&sa=D&ust=1468346325966000&usg=AFQjCNFDNy5fisGIOs7mRSo8o8Lu-e9FXw) (Youtube) is an example of a Micromouse competition.

**Problem Statement:**

The robotic mouse has two runs for this project. In the first run, the robot will explore and map the unknown maze surface and store its maps in its memory with certain time limit. It will continue exploring the space no matter it has reached its goal i.e. center of the maze. Then, in the second run the robotic mouse will exploit what it has learnt in the first run and come up with optimal strategy from start location to the end location and will plan its route in a optimal way.

Some initial code is already provided along with specifications for the robotic mouse environment and testing. I have programmed the robotic mouse to first plot and then optimally navigate through that virtual environment. For the first run the robot will explore as much surface as it can in minimal steps and for then for the second run, it will use artificial intelligence search methods to come up with optimal route to reach center of maze.

**Metrics:**

For scoring the performance of the robotic mouse, combination number of steps taken by the robotic mouse in the first run plus number of steps for the second run. For adding more weight to the second run, the first run has been divided by thirty and then added to the second run steps to make the final score. Total number of steps taken by the robotic mouse are limited to 1000.

# Analysis

**Data Exploration:**

The shape of virtual maze is n\*n square with goal in the center of it, which is 2\*2 grid. For each cell of the grid, it might has walls in the left, right, up or down side of the cell, which will prohibit the robot motion though them. Robot will always start from the bottom left corner of the virtual maze where it can only all walls closed except for the up one.

The robot has three distance sensor mounted on it, that will measure the robot distance from its current location to the walls on their respective sides. It is assumed that the sensor readings are accurate and free from any noise. Below are these three sensors,

* Left distance measurement sensor
* Front distance measurement sensor
* Right distance measurement sensor.

For the robot motion, it can rotate either clockwise or anticlockwise direction or move straight and then move forward or backward. It is assumed the robotic motion is perfect with no probability of moving into the wrong direction. If the robot hits the wall while moving, it will stay where it is after that time step and there is not penalty for this collision. Sensors values are available after each movement of the robot.

Rotation of the robot is expressed in angles with possible values -90(counter clockwise), 0 or 90(clockwise) degrees. And the movement is expresses as an integer with possible values form -3 to 3 inclusive, which indicates the number of steps taken in the respective direction.

Mazes are provided as text files with first lines describing the dimension of the square maze and the subsequent lines indicates the allowed location for each column delimited by commas. Each number represents a four-bit number that has a bit value of 0 if an edge is closed (walled) and 1 if an edge is open (no wall); the 1s register corresponds with the upwards-facing side, the 2s register the right side, the 4s register the bottom side, and the 8s register the left side. For example, the number 10 means that a square is open on the left and right, with walls on top and bottom (0\*1 + 1\*2 + 0\*4 + 1\*8 = 10), as shown below.

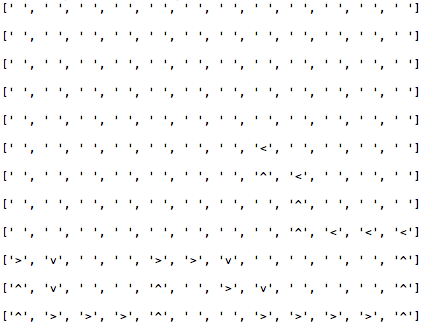
****

These maze files are only used for testing the search approach independently and for testing the robot performance, but the robot will plot the unknown virtual environment by itself, rather than from these files.

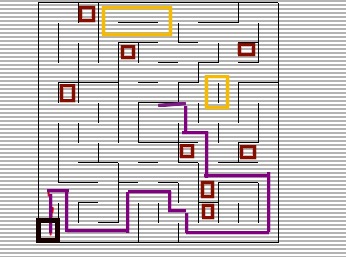
For example for the first maze provided, its visualization will be covered in the next section. Some of the interesting points that can be exploited for obtaining better score are avoiding dead-ends, avoiding loops, using initial heuristics to guide the robot towards the center of maze, and preferring the front motion as compared to turning. They are further discussed in the next section.

**Exploratory Visualization:**

For the first maze, it is shown below with optimal route from bottom left corner to the center of the maze output by the program is shown below, where the arrows indicate the robot moving direction.

****

and the equivalent path plotted on the actual maze is shown below, with thick black rectangle on the bottom letf corner indicating the start position of the robot and violet color lines indicating the robot trajectory.

****

The red rectangular blocks indicating dead-ends in the maze where if the robot can enter leaves no option for further motion until rotate 180 degrees or turn back. Yellow blocks indicate some potential loops that the robot must avoid during its exploration phase to move in those loops. As during the exploration phase the robot will try to maximize the coverage of the space along with finding the optimal route to its goal, so the robot will loose its number of steps while moving in the same loop again and again.

When it comes to the optimal path, although the length of the path is 30, but considering the fact that the robot can move 3 units per move, it can achieve the above drawn path in just 17 moves. So, in the subsequent algorithm, I have preferred forward motion as compared to turning.

**Algorithms and Techniques:**

As there are two runs allowed for the robotic mouse with total number of steps limited to 1000. So, during the first run, I have compared performance following algorithms,

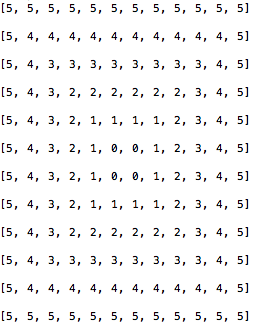
* Random movement,
* Random movement with weighted choice based on the count frequency,
* Weighted choice based on count frequency while avoiding dead ends
* Weighted choice based on count frequency while avoiding dead ends and heuristics guide

The random movement technique will work for small grid size and will reach goal, but its coverage is low as compared to other techniques. But as the grid size increases, its performance start to degrade substantially and will not suffice to reach goal for most of cases.

Now coming to the second approach with weighed choice for the next move, I have counted frequency for each of cell of the maze during its exploration phase. And the robot uses weighted random selection based on assigning low probability to the high frequency cells and coverage is high even for large size mazes. So, it results in better coverage as compared to the previous technique, but it waste a lot of time in dead ends.

For the third technique, I also captured information about the dead-ends, using the fact that all of the three sensors values will be zero for that that location. So, while making next movement, I used that dead-end information captured in the previous movements to avoid trapping in these dead-ends by turning 180 degree. I also removed random selection for this approach and used minimum of all count frequency for all of the possible choices for giving absolute 1 probability value to the location less explored.

For the last technique, I make a initial heuristic function using information available like the center of the maze, by initializing the goal locations with zero heuristic value and then the value gradually increases in the subsequent goals. Sample initial heuristics is shown below,



So, in this technique, instead of just using previous captured frequency and dead end information, I also used this heuristics value to count for deciding next location. And the results of this approach are significantly better than as compared to the previous approaches in the first run.

For the second run, I used A start search method for using mapped information about the maze to output optimal route to the center of the maze using the heuristics value again compiled. Now heuristics values are compiled using the mapped information. Starting from the goal, I counted for each cell the minimum number of steps it takes to reach the goal using optimum policy algorithm. Then using that heuristics value, I have come up with optimal value for the route. This approach will outperform when the coverage of the maze is high. I also assigned more cost value to the left or right turn as compared to the forward motion while computing the G value for the A start search method. By that way, I will come up with more strait lines on which the robot can travel multiple units and save time later.

**Benchmark:**Comparing performance of different techniques using the criteria that 1/30 of the number of steps of first run plus the number of steps in the second run, I compared performance of all of them. The random method performance is worst among all of them, and I used that as my baseline method for further comparing performance of other advanced algorithms used for this project.

# Methodology

**Data Preprocessing:**

Since the environment and robot specifications are given, so not data preprocessing steps are required for that projects. Also sensors data is free from noise, so no noise removal or cleaning steps are required for this project.

**Implementation:**

I implemented all of the required functionality inside the robot.py class. Firstly coming to tracking robot motion inside robot.py class for keeping track of its direction and taking respective actions, I used following constructs for defining its rotation,

forward = [[-1, 0], # go up

[ 0, -1], # go left

[ 1, 0], # go down

[ 0, 1]] # go right

action = [1, 0, -1]

forward\_name = ['up', 'right', 'down', 'left']

angle\_val = [-90, 0,90]

This is used for taking robot to take various directions, like moving up, etc. Furthermore, forward\_name list is used for defining current robot direction. And then the action list is used for mapping corresponding robot rotations to the forward list index. Like if robot want to turn right, then it will decrement its current orientation head by one as in the action list i.e. -1 and its corresponding forward list coordinates will be multiplied with its current coordinates to make its new location.

Then, initialized some variables that will be used for further logic implemented and are documented in the code. Some of them required description, and they are

self.count\_grid = [[0 for row in range(self.no\_of\_rows)]

for col in range(self.no\_of\_cols)]

self.mapped\_grid = [[self.init\_val for row in range(self.no\_of\_rows)]

for col in range(self.no\_of\_cols)]

self.deadend\_grid = [[0 for row in range(self.no\_of\_rows)]

for col in range(self.no\_of\_cols)]

self.heuristics = [[self.def\_heu\_val for row in range(self.no\_of\_rows)]

for col in range(self.no\_of\_cols)]

self.init\_heuristics = [[self.def\_heu\_val for row in range(self.no\_of\_rows)]

for col in range(self.no\_of\_cols)]

self.route = []

**count\_grid** is used for keeping frequency of visit of each cell as the robot explores the space. **mapped\_grid** contain values of the mapped wall position for each cell, and they are computed as follow, Each number represents a four-bit number that has a bit value of 0 if an edge is closed (walled) and 1 if an edge is open (no wall); the 1s register corresponds with the upwards-facing side, the 2s register the right side, the 4s register the bottom side, and the 8s register the left side. For example, the number 10 means that a square is open on the left and right, with walls on top and bottom (0\*1 + 1\*2 + 0\*4 + 1\*8 = 10).

**deadend\_grid** is used for keeping track of dead-ends detected while exploring the mace, **heuristics** contains respective heuristics values developed after the maze is mapped and the robot will use it for calculating optimal path. **init\_heuristics** is used for heuristic exploratory which is made at robot initialization with goal filled with zero values and its value is decremented for each cell after that. And **route** list contains optimal path values in the form of (rotation,movement) for the final optimal path used in the second run of robot.

Then, I used various helper method for updating and fetching values from and to the respective grids. And some other methods like

def **is\_all\_space\_explored**(self):

for i in range(len(self.count\_grid)):

for j in range(len(self.count\_grid[0])):

if self.count\_grid[i][j] ==0:

return False

return True

which will be used for checking whether robot explores all of the maze or not by checking all elements of the count\_grid values, and if any one of them is zero, then it returns true.

def **move**(self, steering, distance):

# make a new copy

curr\_pos = dir\_names.index(self.heading)

act\_index = angle\_val.index(steering)

dir\_index = (curr\_pos+action[act\_index])%len(dir\_names)

self.heading = dir\_names[dir\_index]

delta = forward[dir\_index]

self.location = [ self.location[i]+delta[i]\*distance for i in range(2) ]

and then the move method is used for updating self.**location** which is location of current position of robot and self.**heading** which contains robot current heading. These are updated by taking corresponding steering and movement by the robot.

And then there come other method like

def **weighted\_choice**(self,choices):

values, weights = zip(\*choices)

total = 0

cum\_weights = []

for w in weights:

total += w

cum\_weights.append(total)

x = random() \* total

i = bisect(cum\_weights, x)

return values[i]

which is used for getting weighted probability of the choices based on the values of second factor. I used it for getting possible action based on the weighted probability of their respective visit frequencies.

And coming to the weighted\_prob\_exploration method, which is used to return weighted selection of the possible next location based on the frequency visit values. It assign very low probability to the deadend locations to avoid selecting them.

def **weighted\_prob\_exploration**(self,sensors\_array):

weighted\_array = []

for sensors\_item in sensors\_array:

x,y = self.simulate\_move(angle\_val[sensors\_item],1)

if self.count\_grid[x][y] == 0:

sensor\_weight = self.self\_exploration\_val

else:

if self.deadend\_grid[x][y] == 1:

sensor\_weight = self.epsilon\_val

else:

sensor\_weight = self.self\_exploration\_val \*\*(-self.count\_grid[x][y])

weighted\_array.append((sensors\_item,sensor\_weight))

return self.weighted\_choice(weighted\_array)

Similarly weighted\_prob\_exploration\_heu method is incorporating self.init\_heuristics for making next location decision based on the low value of the total of visit frequency and self.self.init\_heuristics value. And the other two remaining methods counting\_exploration and counting\_exploration\_heuristic are used for the same purpose but they are not probabilistic but rather using minimum value of the total sum.

Two important methods for checking when to stop first run of the robot are

def **all\_space\_explorered**(self):

return self.is\_all\_space\_explored() or self.count\_steps >=900

def **goal\_first\_explorere**(self):

return self.isGoal(self.location) or self.count\_steps >=900

all\_space\_explorered stops the first run, when all of the maze is explored and the second method stops the first run when the robot reached goal. One important thing about that is they both will terminate first run if the maximum number of steps taken by the robot in the first run exceeded from 900. As this will provide enough steps for robot in the second run for reaching goal.

In the **robot\_exploration** methodis used for exploring the maze in the first run. It can use any of the two criteria either goal\_first\_acheived or all\_space\_explored for stopping the first run. And it can also use any of the four next location decision functions for making next location decision. Those four methods are,

* weighted\_prob\_exploration
* weighted\_prob\_exploration\_heu
* counting\_exploration
* counting\_exploration\_heuristic

It also updates the self.mapped\_grid for each of the cell using the sensor values based on the sensor values. In the end of this method it updates robot current location and robot rotation based on the action it has taken. When the robot meets criteria for the first run to terminate, it sets is\_changed\_explorat flat to true for moving onto the second run of the robot.

Now coming to the second run, after the robot has mapped maze based on the criteria, it further build new heuristic values based on the mapped grid of maze. This logic is in **build\_heuristics** function, which

**Problem Statement:**

**The robotic**

**Problem Statement:**

**The robotic**

**Problem Statement:**

**The robotic Problem Statement:**

**The robotic**

**Problem Statement:**

**The robotic**

**Problem Statement:**

**The robotic**

**Problem Statement:**

**The robotic**

**Problem Statement:**

**The robotic**

**Problem Statement:**

**The robotic**

**Problem Statement:**

**The robotic Problem Statement:**

**The robotic**

**Problem Statement:**

**The robotic**

**Problem Statement:**

**The robotic Problem Statement:**

**The robotic**

**Problem Statement:**

**The robotic**

**Problem Statement:**

**The robotic**

**Problem Statement:**

**The robotic**

**Problem Statement:**

**The robotic**