Report

COSC276, Fall2021, PA6, Xuedan Zou

A. Description

We implement the filter algorithem here to solve this robot moving problem based on Hidden Markov Model. In short, we face two challenges here: 1. to implement the color sensor of the robot and the pysical moving rules based on the maze. 2. to implement the filtering process, which includes the transition process and the updating process.

To begin with, the robot just needs to know how many steps it should take. It will then choose the direction to move randomly, and only know it does move without the specific direction. We should write a move function to simulate each random move and add the physical constraint here: if the moving direction has a wall then the robot should stay. We use a **dictionary** here to save all of the possible moving and the change on the location coordinate, we can then change the robot's location coordinate based on this easily without using complex *if....elif...* structure:

```
self.movelist = {"n": (0, 1), "s": (0, -1), "w": (-1, 0), "e": (1, 0)}
......
next_x = self.robotloc[0] + self.movelist[action][0]
next_y = self.robotloc[1] + self.movelist[action][1]
```

Then we should implement the color sensor model based on the given probability rule. We use a random seed here to generate a random number between 0 and 1. Knowing that with only 0.88 possiblity we can get the currect color, it is not hard for us to set this rule to our color sensor. So far we have modeled our problem. We only need to give out the total steps the robot should move, then the robot will move accordingly randomly, following the physical rule and keep getting color data from the color sensor.

We then implement the filtering algorithm. First is the transition part, for each step suppose we know the probabilities whether the robot is on one specific cell from every cell, then according to HMM, we should multiply these probabilities with the probabilities on each "from" cell accordingly. Consider a transition table where n represents the total number of cells of the given maze:

```
    i11 i12 i13 ... i1n
    i21 i22 i23 ... i2n
    ....            ....
    in1 in2 in3 ... inn
```

For each row in this table, the row number represents the "from" cell's index in the maze, and for each column in the table, the column number represents the "to" cell's index in the maze. So for example, i13 reprents the possiblity from the cell with index 1 to the cell with index 3. Since the robot doesn‘t know how it moves, this table is constant after we give out the maze. If it doesn't have wall at all, consider each cell, if that cell is the wall, then the probablity from and to this cell should be 0. If not, the previous cell could come from the north, south, east and west four directions so in the 9-neighbor area, the probability for each cell to be the previous cell of the center cell below is like this:

```
 0    1/4    0
 1/4   0    1/4
 0    1/4    0
```

If a specific direction has a wall, then the probability from that direction will down to 0 and at the same time the probability the robot stays at the same place will increase 1/4 like the images shown below:

```
 0    1/4   0
 1/4  1/4  1/4
 0     #    0

 0     #    0
 1/4  1/2  1/4
 0     #    0

 0     #    0
 1/4  3/4   #
 0     #    0

 0     #    0
 #     1    #
 0     #    0
```

According to this rule, we can generate our transition table according to our given maze. Then consider the matrix multiplication process, suppose we have the probability distribution of the previous step as a matrix where n reprents the total number of cells:

```
(i1, i2, i3, .... in)
```

Then if we de the matrix multiplication using the previous probability distribution matrix and the transition table, we can get a new probability distribution of this step:

```
                          i11 i12 i13 ... i1n
                          i21 i22 i23 ... i2n
(i1, i2, i3, .... in)     ....            ....     = (new1, new2, new3, .... newn)

                          in1 in2 in3 ... inn
```

Next consider our updating part, for each cell there is a 0.88 probability that the robot gets the correct color of that cell, and 0.04 probability that the robot gets the wrong color of that cell if the robot is at that cell. So it is no hard to find that we should compare the sensor's color with each cell's original color, if it is the same we then multiply its probability value in the new probablity distribution matrix with 0.88. If not, we multiply it with 0.04. We then do the normalization process by dividing each cell's possiblity to the sum of all possiblities. So then we finish a filtering step and get the new probablity distribution matrix. Loop this process to get the final step's probablity distribution matrix. Specially, the probablity distribution matrix in the beginning (before step 1) is: for each floor cell in the maze the probablity should be 1/total_number_of_floors, and for each wall cell in the maze the probablity is 0.

Addtionally, to better implement the matrix muliplication process here, we use *numpy* library here. To create a matrix in numpy, we use code like this to get matrix in numpy where probability_list is a list:

```
import autograd.numpy as np
np.array(self.probability_list)
```

Also we do the matrix multiplication like this in numpy:

```
 self.probability_list = np.dot(self.probability_list, self.transition_table)
```

In order to change the value inside the ndarray, we first convert the data into list and modify it, then convert it to the ndarray agian.

```
        self.probability_list.tolist()
        # update
        for x in range(0, self.maze.width):
            for y in range(0, self.maze.height):
                if self.maze.get_color(x, y) == self.color_list[-1] :
                    self.probability_list[self.maze.index(x, y)] *= 0.88
                else:
                    self.probability_list[self.maze.index(x ,y)] *= 0.04

        np.array(self.probability_list)
```

B. Evaluation

We test our code on multiple test cases and in general we believe our code works fine. We notice that in some specific cases, no matter how many steps the robot has taken, once the color sensor goes wrong in the last step, the probability distribution table will be destroyed. This makes sense as the HMM will take a few steps to fix any sensor error, see below:

```
step:  18
try to move: w
ggyyby
#ry##r
ygrbyy
rg##ry
gbbr#b
bRrgrg

the color sequence the robot has got so far is:
['b', 'b', 'g', 'b', 'g', 'g', 'b', 'g', 'b', 'b', 'r', 'r', 'r', 'b', 'r', 'g', 'r', 'g']

0.000005  0.000287  0.000000  0.000000  0.000000  0.000000

0.000000  0.000020  0.000025  0.000000  0.000000  0.000000

0.000336  0.000745  0.000019  0.000012  0.000001  0.000000

0.000344  0.011294  0.000000  0.000000  0.000002  0.000008

0.014438  0.000200  0.017632  0.015108  0.000000  0.000028

0.000458  0.231400  0.010546  0.536500  0.013619  0.146972


step:  19
try to move: w
ggyyby
#ry##r
ygrbyy
rg##ry
gbbr#b
Rgrgrg

the color sequence the robot has got so far is:
['b', 'b', 'g', 'b', 'g', 'g', 'b', 'g', 'b', 'b', 'r', 'r', 'r', 'b', 'r', 'g', 'r', 'g',
'g']
```

```
0.000232  0.000241  0.000011  0.000000  0.000000  0.000000

0.000000  0.000038  0.000002  0.000000  0.000000  0.000000

0.000062  0.009011  0.000028  0.000002  0.000001  0.000000

0.000927  0.009716  0.000000  0.000000  0.000000  0.000001

0.011922  0.009644  0.001526  0.020509  0.000000  0.005161

0.008661  0.187328  0.027941  0.444584  0.024944  0.237508


step:  20
try to move: w
ggyyby
#ry##r
ygrbyy
rg##ry
gbbr#b
Rgrgrg

the color sequence the robot has got so far is:
['b', 'b', 'g', 'b', 'g', 'g', 'b', 'g', 'b', 'b', 'r', 'r', 'r', 'b', 'r', 'g', 'r', 'g',
'g', 'b']

0.000048  0.000027  0.000013  0.000001  0.000000  0.000000

0.000000  0.000478  0.000004  0.000000  0.000000  0.000000

0.000518  0.000507  0.000466  0.000036  0.000000  0.000000

0.001165  0.001508  0.000000  0.000000  0.000000  0.000266

0.001604  0.238390  0.067521  0.025077  0.000000  0.280677

0.245274  0.012024  0.034047  0.026665  0.037682  0.026003
```

Our probablitiy distribution matrix works nice in step 18. However, in step19 the robot detects wrong color and apprently our distribution matrix is broken. In step 20 the color sensor gets the correct color and that distribution matrix is partly fixed.

We also test some interesting test case like below where the robot can just stay at the same place:

```
step:  20
```

```
try to move: n
####
####
#R##
####

the color sequence the robot has got so far is:
['r', 'g', 'r', 'y', 'r', 'r', 'r', 'r', 'r', 'r', 'r', 'r', 'r', 'r', 'r', 'r', 'r', 'r',
'r', 'r']

0.000000  0.000000  0.000000  0.000000

0.000000  0.000000  0.000000  0.000000

0.000000  1.000000  0.000000  0.000000

0.000000  0.000000  0.000000  0.000000
```

and the situation when robot can just move in a line:

```
step:  20
try to move: n
####
####
rRby
####

the color sequence the robot has got so far is:
['r', 'r', 'g', 'r', 'b', 'g', 'r', 'r', 'r', 'r', 'g', 'g', 'r', 'r', 'r', 'r', 'r', 'r',
'g', 'g']

0.000000  0.000000  0.000000  0.000000

0.000000  0.000000  0.000000  0.000000

0.028348  0.951340  0.020296  0.000015

0.000000  0.000000  0.000000  0.000000
```