report.md

Hidden Markov Models

Assignment 6
CS 276, Artificial Intelligence
Fall 2021, Dartmouth College
Spencer Bertsch

This report contains two sections:

1. Description
2. Evaluation

Description

1. How do your implemented algorithms work?

This problem uses a Hidden Markov Model to approximate the location of a robot as it takes random actions (N, S, E, W) across an n x n board. The board squares are each colored either red, yellow, green, or blue, and the robot's only sensor is a color sensor pointing down that is not always quite accurate. However! The robot knows what the board looks like before it starts, but it doesn't know where it's starting from. The job of the filtering algorithm is to generate a probability distribution that eventually finds the most probable position of the robot simply given its transitions between colors as the robot moves across the board.

The implemented algorithm is a filtering algorithm which takes a series of sensor readings - *E(t)* - and produces a probability matrix representing the distribution representing the likely locations of the robot for a certain time (t).

There are two important components to this problem:

1. The Sensor Model
2. The Transition Model

The Sensor Model

In this case the sensor model is a little different than the implementation we saw in the textbook because we have a robot that could be in any one of *n* positions on a board. In order to generate a correct sensor model, we need to understand the performance of the sensor being used to detect the colors of the board. At each time step (t), the robot moves (might hit a wall and stay in the same position) and a color is read by the sensor. After the color is read, then the sensor model can be generated in the filtering algorithm by giving a 0.88 probability to the colors in the map that match the color that was just sensed, and a 0.04 probability to all other colors. This reflects the performance of the sensor attached to the robot.

For example, if the robot were in the top left position of the following board:

```
['B', 'G', 'R', 'Y']
['Y', 'B', 'R', 'R']
['R', 'Y', 'G', 'R']
['B', 'G', 'G', 'B']
```

Then the sensor reading would likely be Blue. In that case, the Sensor Model would look like this:

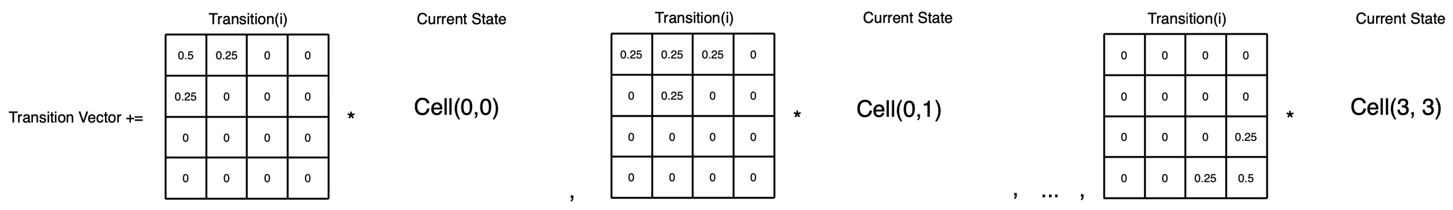
```
[0.88, 0.04, 0.04, 0.04]
[0.04, 0.88, 0.04, 0.04]
```

```
[0.04, 0.04, 0.04, 0.04]
[0.88, 0.04, 0.04, 0.88]
```

The sensor model reflects an equal probability of the robot being in any of the blue cells.

The Transition Model

The transition model is the heart of the filtering algorithm. In order to produce the transition vector at each iteration, we loop through each cell in the maze and multiply a matrix representation of the likelyhood of the robot's new position given random movement *if* the robot were in that cell by the scalar value of the *Current State* for that given cell value. See the Filtering_Diagram.pdf in the `docs` directory for a better description.



In short, we take the cumulative sum of each of the individual transition matrices multiplied by the current state at each of the respective cells in the maze. This cumulative sum is what gives us the transition vector! This transition vector (or transition model) is extremely important because we then use it to compute the new *Current State*.

How each distribution of states is computed

In order to understand the algorithm fully myself, I laid out the pseudocode visually. At first I didn't understand what values needed to be represented as matrices and what values should be scalars, so I made a Draw.io diagram of the algorithm logic. Please see the PDF in the `docs` directory for a high
resolution visual representation of the filtering algorithm. In addition to the visual representation of the algorithm, I will also go through each step here.

The distribution of states is computed using the filtering algorithm that takes a list of sensor readings as input. As described above in the *Sensor Model* and *Transition Model* sections, we first compute the sensor model matrix which represents a 0.88 probability for each of the positions that match the color that was just sensed, and 0.04 everywhere else. We then create the transition model in which we take the cumulative sum of each of the transition matrices multiplied byt eh *Current State* at each cell.

After this, we multiply the prediction vector and the transition vector using element-wise multiplication using the Numpy python scientific computing library to generate a new prediction vector. At this point we're almost done, we just need to normalize the new prediction vector. This is needed because we want a matrix of probabilities, but at this point the sum of the matrix elements will likely not equal one. I do this by simply summing all the values in the matrix and then dividing each element by the sum. And that's it! Here we can print the ground truth to see where the robot is, and we can also print the *Current State* to see how close our distribution matches the ground truth.

What design decisions did you make?

I created a HMM (hidden markov model) class, and I placed all the methods that I needed to initialize the maze inside that class. I also implemented the Filtering algorithm as a method of that class and used several helpful instance variables such as the transition vector (which I pre-computer once to save time) and the sensor readings, ground truth state, robot path, etc.

I also made the design decision to create heatmaps in order to view the performance of the *Current State* compared to the ground truth location of the robot in the maze. See the end of this report to find some examples of how performant the *Current State* at time steps t=0, t=3, and t=9.

Evaluation

After much bug fixing, the filtering algorithm is working exactly as expected. In addition to the printed output, I also used matplotlib to generate some heatmaps showing how similar the *Current State* is to the actual ground truth location of the robot at time (t). In addition to the heatmaps, there is of course also the printed output showing the performance of the filtering algorithm and the current state at each time step. Example output can be seen below:

```
 Output for Time t=12
----- Prediction Vector -----
[0.04, 0.88, 0.04, 0.04]
[0.04, 0.04, 0.04, 0.04]
[0.04, 0.04, 0.88, 0.04]
[0.04, 0.88, 0.88, 0.04]
----- Ground Truth: X12 -----
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 1, 0, 0]
----- Current State -----
[0.00191096 0.01706248 0.00039611 0.00021325]
[1.82056894e-03 1.99524254e-02 4.24476130e-04 3.93714434e-05]
[2.08457428e-02 1.27343151e-03 4.25105696e-01 1.11806422e-04]
[0.00297936 0.46610947 0.04080118 0.00095368]
```

We can see from the above output that the *Current State* at time t=12 has a probability of 0.466 for the robot being in location (3, 1) modeled where the origin is in the upper left corner. Sometimes the sensor produces a faulty reading, causing the *Current State* to be inaccurate, but after another one or two time steps with accurate sensor readings the state reflects the true location of the robot again.

See below for an example run showing heatmaps for the ground truth and the *Current State* at times t=0, t=3, and t=9. As time progresses, the state becomes more and more accurate, yielding a higher and higher probability for where the robot actually exists in the maze. As mentioned before, if the sensor were to produce a faulty reading for a time step, then the distribution of the *Current State* might seem like a poor model for that time step, but the accuracy of the state is remedied after one or two accurate sensor readings.

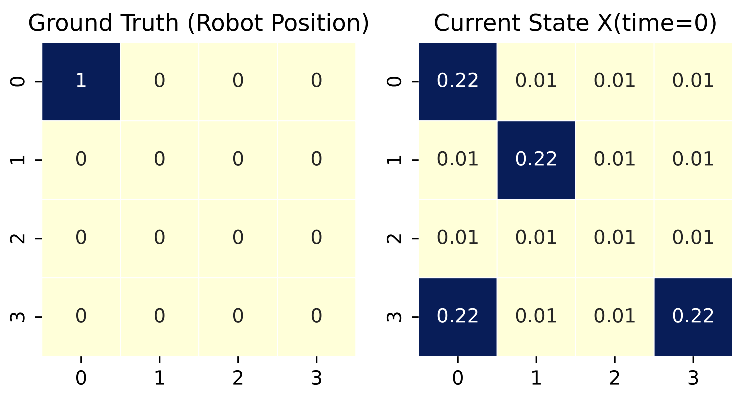


Figure 1: Ground truth function of the starting state with the robot in cell (0, 0). The state refelcts an equal probability of the robot in any of the states that it sees are blue.

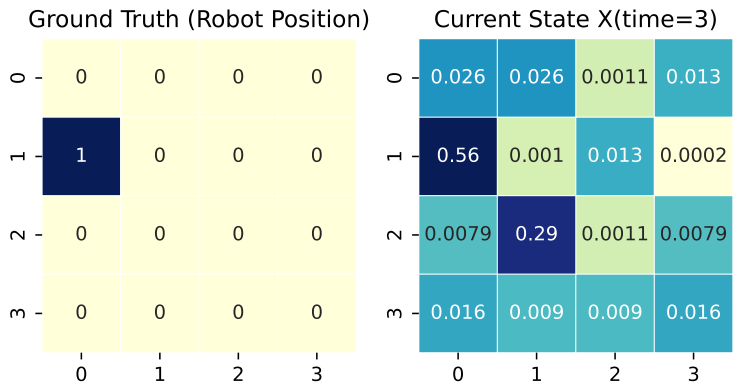


Figure 2: Ground truth function of state at time t=3 with the robot in cell (1, 0). After only three moves, we can already see that the Current state lists the true location as the most probable, but we can get better!

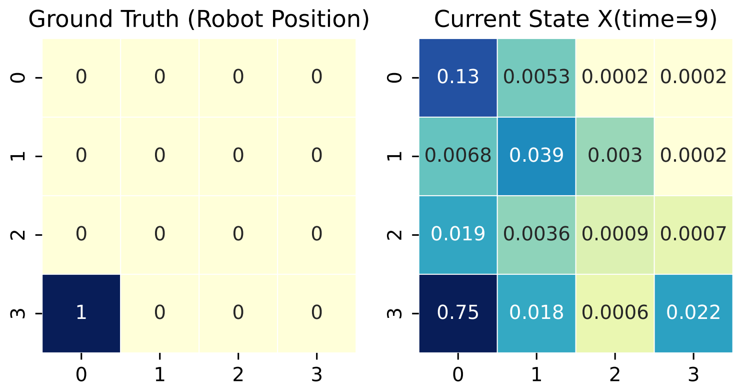


Figure 3: Ground truth function of state at time t=9 with the robot in cell (3, 0). Here we can see that after 9 iterations we get a better result in which the state matrix shows a probability of the ground truth is 0.75