Hidden Markov Model

In this study, I tackle the problem of estimating a robot's position within a nxn maze, using a Hidden Markov Model (HMM) framework. I developed an algorithm that employs probabilistic filtering to deduce the robot's location based solely on sensory input, in the form of color signals corresponding to different sections of the maze. Without prior knowledge of its starting point or the directions moved, my model calculates the transition probabilities between states and adjusts for sensor errors, leading to a real-time probability distribution for the robot's various possible locations. This report documents the precise construction of my model, the steps taken in the algorithm's implementation, and the method's effectiveness through comparison with the robot's actual trajectory.

Implementation

- **Maze and Robot classes**

  I began by defining two essential classes: `Maze` and `Robot`. The `Maze` class encapsulates the maze structure and functionalities such as maze generation and wall insertion. To represent the maze, I utilized a 2D list where each cell is initialized to a random color denoted by `'R'`, `'G'`, `'B'`, `'Y'`. The `Robot` class is designed to maintain the belief state and models for state transitions and sensor readings.

- **Maze generation and walls**

  In the `Maze` class, I implemented the `generate_rand_maze()` method to create a maze of given dimensions with randomly assigned colors. The walls are placed using the `insert_wall_in_maze()` method, allowing for a flexible maze configuration. This configuration is printed in ASCII art for a visual representation of the maze.

- **State transition model**

  The `generate_transition_model()` method in the `Robot` class populates the transition model based on possible movements (North, South, East, West). I considered the constraints imposed by walls, ensuring the robot's position stays the same if a movement is blocked.

- **Sensor model**

  I developed the `generate_sensor_model()` to reflect the probability of the sensor's accuracy. The robot's downwards sensor reads the color of the current square with an `88%` chance of accuracy, while there's a `4%` chance for each of the other colors, accounting for possible sensor errors.

- **Filtering method**

  The core of my solution lies in the `predict` and `update` methods, which together implement the filtering algorithm. The predict method calculates a new belief state by considering the probabilities

of moving from one state to another. The update method adjusts these beliefs based on the sensor input. By normalizing the probabilities after the update step, I ensure that the belief state remains a probability distribution.

  - `predict`

    In the `predict()` function, for each state, I iterated over all possible actions and updated the new belief state based on the transition model probabilities. This represents the prediction phase of the filtering process, where we consider the effect of the robot's actions on its location probabilities without yet accounting for the new sensor information.

  - `update`

    During the update phase, implemented in the `update()` method, the belief state is modified by the likelihood of the sensor reading in each state. If the robot senses 'R' on a red square, the belief for being on that square is multiplied by 0.88, reflecting the sensor model.

  By executing the `filter()` function, which calls `predict` and `update` in sequence for each sensor reading, I generate and output the sequence of belief states. I also implemented a function to display the most probable location of the robot at each step, providing a clear and immediate understanding of the algorithm's performance.

- **Execution**

  Finally, the sensor_read method takes a sequence of sensor readings and processes them through the filtering method, outputting the belief states and the most likely position of the robot at each time step. This not only demonstrates the algorithm's real-time processing capabilities but also validates its accuracy by comparing it with the actual robot trajectory.

Result

```
input sequence = ['Y', 'B', 'Y', 'G']

distribution in step 0:
(1, 1) -> 0.008547008547008544
(1, 2) -> 0.18803418803418798
(1, 4) -> 0.18803418803418798
(2, 1) -> 0.18803418803418798
(2, 4) -> 0.008547008547008544
(3, 1) -> 0.008547008547008544
(3, 2) -> 0.18803418803418798
(3, 3) -> 0.008547008547008544
(3, 4) -> 0.008547008547008544
(4, 1) -> 0.18803418803418798
(4, 3) -> 0.008547008547008544
(4, 4) -> 0.008547008547008544
```

```
distribution in step 1:
(1, 1) -> 0.495349975526187
(1, 2) -> 0.03279490944689182
(1, 4) -> 0.03279490944689182
(2, 1) -> 0.022515907978463045
(2, 4) -> 0.2692119432207538
(3, 1) -> 0.03279490944689182
(3, 2) -> 0.022515907978463045
(3, 3) -> 0.012236906510034264
(3, 4) -> 0.0019579050416054823
(4, 1) -> 0.03279490944689182
(4, 3) -> 0.0019579050416054823
(4, 4) -> 0.043073910915320604

distribution in step 2:
(1, 1) -> 0.025593120875699118
(1, 2) -> 0.3195966418759506
(1, 4) -> 0.19787063318123577
(2, 1) -> 0.30853064108552203
(2, 4) -> 0.014024120049341915
(3, 1) -> 0.0027066192409489938
(3, 2) -> 0.04847962251044923
(3, 3) -> 0.0009461191151989845
(3, 4) -> 0.007988119618199023
(4, 1) -> 0.07061162409130649
(4, 3) -> 0.0014491191511275586
(4, 4) -> 0.00220361920502042

distribution in step 3:
(1, 1) -> 0.05210698826348007
(1, 2) -> 0.075507455666527
(1, 4) -> 0.04660893945072792
(2, 1) -> 0.049502649352176416
(2, 4) -> 0.017941920027044703
(3, 1) -> 0.7261870641062383
(3, 2) -> 0.0077174783752599705
(3, 3) -> 0.004515106084323624
(3, 4) -> 0.0019300585723629575
(4, 1) -> 0.016456482277634465
(4, 3) -> 0.0004639122222957131
(4, 4) -> 0.0010619456019284048
(1, 1) = 0.0521069882634801
(1, 2) = 0.0755074556665270
(1, 4) = 0.0466089394507279
(2, 1) = 0.0495026493521764
(2, 4) = 0.0179419200270447
(3, 1) = 0.7261870641062383
(3, 2) = 0.0077174783752600
(3, 3) = 0.0045151060843236
(3, 4) = 0.0019300585723630
```

```
(4, 1) = 0.0164564822776345
(4, 3) = 0.0004639122222957
(4, 4) = 0.0010619456019284

problem:       Prediction:

# # # # # #    # # # # # #
# B Y # Y #    # B Y # Y #
# Y # # B #    # Y # # B #
# G Y R R #    # ● Y R R #
# Y # R B #    # Y # R B #
# # # # # #    # # # # # #

most likely position 👉  (row: 3, col: 1) = 0.7261870641062383`
```