Probabilistic Reasoning

Vivek Sah

February 29, 2016

1 Introduction

In this paper, we are going to implement probabilistic reasoning over time in the context of a blind robot which uses probabilistic models to determine its location in a provided maze. We will implement filtering algorithm as described in "Artificial Intelligence" by Russell and Norvig to estimate the position of the robot. In a filtering algorithm, we need to maintain a current state estimate and update it whenever we get an evidence. In other words, given the result of filtering up to time t, the agent needs to compute the result for $t + 1$ from the new evidence $e^t + 1$.

2 Overview of Project Structure

In this project, we will implement these classes: *Robot, Sensor, Probabilistic-Model, and SensorProblem* . The SensorProblem class defines and initializes the sensor problem using the other remaining classes. We will look into further implementation details of each of these classes below.

2.1 Robot

The Robot class defines the blind robot with a color-detecting sensor pointing downwards. It has an initial location and will move in the maze according to the sequence of moves provided. However, it does not know its actual location with absolute certainty. We will provide it random sequence of moves and compute the probability distribution of the robot's location in the maze. We will use this distribution to determine the location of the robot in the maze.

In *Robot's* initialization method, it takes a maze and an initial location and stores this information. It also initializes a sensor with the maze provided. The Robot class has two methods: *move* and *getSensorInput.* The *move* method takes an action and updates the position of the robot if the move is legal.

The *getSensorInput* method invokes the already initialized sensor's *getColor* method. Further discussion of sensor class follows. We also save the robot's previous states in the form of $actionX, actionY, colorDetected$ in a linkedList.

2.2 Sensor

The blind robot is equipped with a sensor which detects color of the square of the maze it currently is on. The sensor detects the right color with a probability of 0.88. For example, if the robot is in a blue square, the probability of receiving the evidence "b" is .88, but the sensor might also give the evidence "r" (with probability .04), "g" (probability .04), or "y" (probability .04). The situation for squares with other colors is symmetric. Here's a implementation of *getColor* method.

Listing 1: getColor *Sensor.java*

```
public int getColor(int x, int y){
  int realColorIndex =
      Character.getNumericValue(maze.getChar(x,y));
   if(random.nextFloat()< 0.88){
       return COLORS[realColorIndex];
   }else {
      int randomIndex = random.nextInt(4);
      return randomIndex == realColorIndex?
          COLORS[((randomIndex+1 )% 4)] : COLORS[randomIndex];
  }
}
```

The Sensor class takes a maze to initialize and has one method- *getColor*. In this method, it gets the right color from the maze corresponding to the given location, and with a probability of 0.88, returns the right color; otherwise, returns any of the other color.

2.3 Maze

In this project, the maze has colored squares represented as numbers : {blue: 0, red:1, green: 2, yellow:3}. The maze file will have numbers representing the colors. The maze class reads in a maze file and stores the information in a 2D grid. It has two methods: *isLegal(move)* and *getChar(position)*. The *isLegal(move)* determines if the move is legal by checking if the move results in collision with a wall. The *getChar(position)* returns the number representing the color at the given position.

2.4 ProbabilisticModel

We initialize this class using the probability map from the SensorProblem class.In this class, we will define the transition state and compute probability distribution by implementing filtering algorithm. We have two helper methods in this class: *getTransitionProb(pos)* and *getSensorProb(color,pos)*. getTransitionPro(pos) calculates the probability of transitioning into that position from other positions using the current probability distribution map. Let's look at the implementation in more detail.

Listing 2: getTransitionProb *ProbabilisticModel.java*

```
public double getTransitionProb( int x, int y){
    double probTransition = 0; //probability transition into that
        position
    double probNoTrasition = 0; //probability it stayed where it
        was

    //check if it could have transitioned from the four sides
    int[][] possibleTransitions =
        {{x,y+1},{x,y-1},{x+1,y},{x-1,y}};
    for(int[] possibleTransition: possibleTransitions){
        if(maze.isLegal(possibleTransition[0],possibleTransition[1])){
            probTransition+=0.25*pMap[possibleTransition[1]][possibleTransition[0]];
        } else {
            probNoTrasition+=0.25 * pMap[y][x];
        }
    }
    return probTransition+probNoTrasition;
}
```

Next, the getSensorProb(color,pos) calculates the probability that the sensor gave the correct value. We combine these two functions to implement filtering algorithm in the method - *doFilterPosition*. For first order Markov assumption, we just use the robot's latest position and the latest color detected to determine the probability distribution for a location whereas in the second order Markov assumption, we use current position, color and the previous position, color to calculate the probability distribution for a given position. Let's look at the implementation below:

Listing 3: doFilterPosition *ProbabilisticModel.java*

```
public double doFilterPosition(int x, int y, ArrayList<int[]>
    prevStates, int order){
    double prob = 0.0;
    if (order ==1){
        return getTransitionProb(x,y) *
            getSensorProb((prevStates.get(0)[2]),x,y);
    }else {
        if(prevStates.size() == 1){ //if first order
            return getTransitionProb(x,y) *
                getSensorProb((prevStates.get(0)[2]),x,y);
        } else { //if second order
            int x1; int y1;
            int col;
            x1= x-prevStates.get(0)[0];
            y1= y-prevStates.get(0)[1];
            col = prevStates.get(1)[2];
            if(maze.isLegal(x1,y1)){ //if legal, then use the
                second order or just use the first order
                return getTransitionProb(x1,y1) *
                    getSensorProb(col,x1,y1) *
                    getTransitionProb(x,y) *
                    getSensorProb((prevStates.get(0)[2]),x,y) ;
            }
            return getTransitionProb(x,y) *
                getSensorProb((prevStates.get(0)[2]),x,y);
        }
    }
}
```

3 Sensor Problem

This is the class where we define and initialize the problem. In the main function, we initialize the problem by passing a maze file and an initial location and then call its start function with a parameter of "maxIteration". In the start function, we start a while loop which calls *doMove* method each time. Let's look at the implementation.

Listing 4: doMove *SensorModel.java*

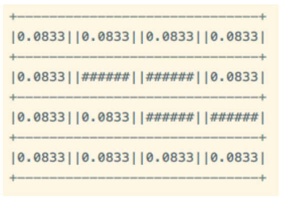
```java
public void doMove(boolean move){
    int[] action = {0,0};
    if (move) {
        action = getNextMove();
    }
    robot.move(action);
    System.out.println("Robot is at x:" + robot.pos[0] + "Robot
        is at y:" + robot.pos[1]);
    int color = robot.getSensorInput();
    int[] currentState = {action[0],action[1],color};
    robot.addToHistory(currentState);
    System.out.println("color sensed is:" + COLORS[color]);
    String Actualcolor =
        COLORS[Character.getNumericValue(maze.getChar(robot.pos[0],robot.pos[1]))];
    System.out.println("actual color is:" + Actualcolor + " ");
    doFilter(robot.getLastTwoStates());
    model.setpMap(probabilityMap);
    printMap();
}
```

In this code, we generate a random action and make the robot to move with that action. Then we get the sensor value(color) and update the robot's "prevMoves" with an array(action[0], action[1], colorDetected). After that we invoke the filtering algorithm which updates the probability map according to the position and color detected by the robot.

4 Results

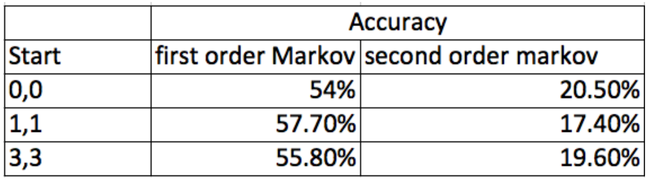
I used the given maze $blue = 0, red = 1, green = 2, yellow = 3$ *simple.maz*
1123
0##0
22##
0132

Here's is the initial probability distribution.



```
+--------------------------------+
|0.0833||0.0833||0.0833||0.0833|
+--------------------------------+
|0.0833||######||######||0.0833|
+--------------------------------+
|0.0833||0.0833||######||######|
+--------------------------------+
|0.0833||0.0833||0.0833||0.0833|
+--------------------------------+
```

Before discussing the results, let us first discuss how we are going to measure accuracy. After every move, we are going to check if the actual position of the robot has the highest probability. If so, we will increase the counter.Now, let's look at the results with different starting points.



	Accuracy	
Start	first order Markov	second order markov
0,0	54%	20.50%
1,1	57.70%	17.40%
3,3	55.80%	19.60%

We can see that the accuracy is much better in first order markov assumption. The reason could be because when the robot gets stuck, the probability of that position skyrockets in few turns, which makes it difficult for other locations to have a higher probability even if the robot is at that location.