Probabilistic Reasing over time

Mengjia Kong

March 8, 2014

1 Introduction

In assignment 2, Mazeworld, we implemented a blind robot with Pacman physics in a maze with many belief states. In that approach, belief states are defined in terms of which world states were possible, but could not tell about which states were likely or unlikely. In this assignment, we will use probability theory to quantify the degree of belief in elements of the belief state. That is what I will discuss about in this report, Probabilistic Reasoning over time.

Probabilistic reasoning over time is an approach that uses a belief state with probability to represent states of the world and a transition model that can predict how the world might evolve in the next time step. From the percepts observed and a sensor model, the agent can update the belief state.

So there are 4 important components in the approach:

Belief state Some unobservable variables decide the current state of the world. Like the textbook, I also use X_t to denote the set of state variables at time t.

Observation Some observable variables can give us some clues about the states of the world, from which we can infer the current states. Like the textbook, I also assume it as E_t to denote the set of observable evidence variables at time t.

Transition model Transition model specifies the probability distribution over the latest state variables, given the previous values, $P(X_t|X_{0:t-1})$. That describs how a state changs to another state with probability.

Sensor model In sensor model, the evidence variables E_t could depend on previous variables as well as the current state variables. We get $P(E_t|X_{0:t}, E_{0:t-1})$ as sensor model.

While the set of state and evidence variables describe the problem, transition model specifies how the world evolves and sensor model specifies how the evidence variables get their values from states.

Having set up the structure of a generic temporal model, there are some basic inference tasks: Filtering, Prediction, Smoothing, Mostlikely explanation and Learning.

Filtering The task is to compute the belief state, the posterior distribution over the most recent state, given all evidence to date. It is $P(X_t|e_{1:t})$.

Prediction The task is to compute the posterior distribution over the future state, given all evidence to date. It is $P(X_{t+k}|e_{1:t})$.

Smoothing The task is to compute the posterior distribution over a past state, given all evidence up to the present. It is $P(X_k|e_{1:t}), 0 \leq k < t$.

Most likely explanation The task is to find the sequence of states that is most likely to have generated those observations. It is to compute $argmax_{X_{1:t}} P(X_{1:t}|e_{1:t})$.

Learning The task is to learn from observations for updating the model. Then the updated models provide new estimates and the process iterates to convergence.

Then I will introduce my implementation of probabilistic reasoning over time by the specific problem, blind robot in a maze. In the next section, I will focus on filtering.

2 Implementation of the model

The implementation is about the specific problem, blind robot problem. In this case, the task is to achiece a sequence of probability distributions describing the possible locations of the robot at each time step, given a sequence of sensor reading, knowledge of the maze, and the colors of each location of the maze.

I create a new class as `BlindRobotMazeProblem` to implement the problem. Here are components in the class:

```
1  private static int actions[][] = {Maze.NORTH, Maze.EAST, Maze.SOUTH, Maze.WEST};
2
3  private Maze maze;
4
5  // transition model P(Lt|Lt-1) = probability
6  // key: ArrayList<Integer>{Lt-1.x, Lt-1.y, Lt.x, Lt.y} value: probability
7  private HashMap<ArrayList<Integer>, Double> transModel;
```

I only store the maze and the transition model in the class and set sensor model as a function, which will be explained later. The construct function is blew:

```
1  public BlindRobotMazeProblem(Maze m) {
2
3    maze = m;
4
5    // build transition model
6    buildTransitionModel();
7
8  }
```

2.1 State variables

What we want to know in the blind robot problem is where the robot is in the maze. Also the location of robot is unobservable. So the state variable is the location with two attributes, (x, y), I set it as L_t representing the location of the robot at time t. The value of the location variable can be every legal location in the maze: all locations except walls in the maze. For example, in the Figure 1, the light grey squares are walls, other red, green, blue, yellow squares are the values of L_t: $(0,0), (1,0), (2,0), (3,0), (0,1), (1,1), (0,2), (3,2), (0,3), (1,3), (2,3), (3,3)$. The initial probability of all these legal location is $1/12$, such as

$$P(L_0 == (0,0)) = 1/12$$

2.2 Evidence variables

What we can infer from to get the state and is observable, is the color that we get from the sensor of robot. I assume the sensor variable as C_t for the color the robot get from its sensor at time t. In this problem, there are only 4 colors, red, green, blue, yellow. They are the values of C_t.

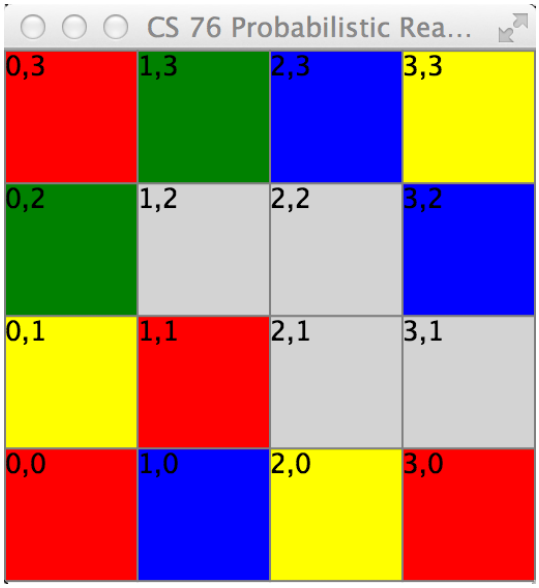

Figure 1: An example of Maze

2.3 Transition model

In general, transition model is $P(X_t|X_{0:t-1})$. Replacing the general variables with speific variables in this problem, the transition model for the blind robot problem is $P(L_t|L_{0:t-1})$. I solve the problem by making a *Markov assumption*: the current state depends on only a finite fixed number of previous states. And I use the simplest one, *first-order Markov process*, in which the current state depends only on the previous state and not on any earlier states. Finally, the transition model is

$$P(L_t|L_{0:t-1}) = P(L_t|L_{t-1})$$

In the maze in Figure 1, using location (0,0) as an example, there are for motions the robot can do: north, south, east, west. If the robot goes a step, we get

$$North : P(L_t == (0,1)|L_{t-1} == (0,0)) = 0.25$$

$$West : P(L_t == (1,0)|L_{t-1} == (0,0)) = 0.25$$

$$South, East : P(L_t == (0,0)|L_{t-1} == (0,0)) = 0.5$$

In my method, I use a `HashMap` to store the transition model. Because it will be used for many times and it is unreasonable to check the maze every time I want to get transition model. Here is my code for transition model:

```
1  // transition model P(Lt|Lt-1) = probability
2  // key: ArrayList<Integer>{Lt-1.x, Lt-1.y, Lt.x, Lt.y} value: probability
3  private HashMap<ArrayList<Integer>, Double> transModel;
```

I use an ArrayList$[L_{t-1}.x, L_{t-1}.y, L_t.x, L_t.y]$ to look up the transition model for $P(L_t|L_{t-1})$. In the problem, not every location is related with else because a robot has only 4 actions to choose. So only probability that is not zero is stored in the `HashMap`. If the robot knocks a wall, it is stuck in the previous location. Here is my code for building the transition model:

```
private void buildTransitionModel() {
  transModel = new HashMap<ArrayList<Integer>, Double>();

  for (int i = 0; i < maze.height; i++)
    for (int j = 0; j < maze.width; j++) {
      //Location (j,i)
      for (int[] action: actions) {
        int curXNew = j + action[0];
        int curYNew = i + action[1];

        // if the action can be done, P(L(curXnew, curYNew)|L(j,i)) = 0.25
        //   else P(L(j,i)|L(j,i)) += 0.25
        if (maze.isLegal(curXNew, curYNew)) {
          ArrayList<Integer> trans = new ArrayList<Integer>();
          trans.add(j);
          trans.add(i);
          trans.add(curXNew);
          trans.add(curYNew);
          transModel.put(trans, 0.25);
        }
        else {
          ArrayList<Integer> trans = new ArrayList<Integer>();
          trans.add(j);
          trans.add(i);
          trans.add(j);
          trans.add(i);
          if (transModel.containsKey(trans))
            transModel.put(trans, transModel.remove(trans) + 0.25);
          else
            transModel.put(trans, 0.25);
        }
      }
    }
}
```

The `isLegal` in `Maze.java` is changed because of the data file of maze is different. Here is the code:

```
// is the location x, y on the map, and also a legal floor tile (not a wall)?
public boolean isLegal(int x, int y) {
  // on the map
  if(x >= 0 && x < width && y >= 0 && y < height) {
    // and it's a floor tile, not a wall tile:
    return getChar(x, y) != '#';
  }
  return false;
}
```

2.4 Sensor model

By replacing the general variables with specific variables in this problem, the sensor model is changed to $P(C_t|L_{0:t}, C_{0:t-1})$. According to the textbook, I make a *sensor Markov assumption* as follows:

$$P(C_t|L_{0:t}, C_{0:t-1}) = P(C_t|L_t)$$

The requirement in this assignment gives us the sensor model for blind robot problem: the probability of receiving the correct evidence of color is 0.88; the probability of receiving the wrong evidence of color for each 3 wrong color is 0.04. Use location $(0, 0)$ as an example.

$$P(C_t == r | L_t == (0,0)) = 0.88$$

$$P(C_t == g | L_t == (0,0)) = 0.04$$

$$P(C_t == b | L_t == (0,0)) = 0.04$$

$$P(C_t == y | L_t == (0,0)) = 0.04$$

In my method, I implement the sensor model in a function. Everytime, I call the function to get the sensor model. Because the sensor table is too large. There are many locations as value of state variable and every state value has at 4 colors the sensor may get. But there are some rules in sensor model. So a function to read sensor model is more reasonable. Here is my code for sensor model:

```
// get P(C==s|L(x,y)) from sensor model
// sensor model: if c(x,y) == s, p = 0.88; else p = 0.04
private double sensorModel(int x, int y, int s) {
  int color = maze.getColor(x, y);

  if (color == s)
    return 0.88;
  else
    return 0.04;
}
```

The `getColor` function is implemented in `Maze.java`, similar to the `isLegal` function. Here is the code.

```
// return the color
// r:1 g:2 b:3 y:4 #:0 illegal:0
public int getColor(int x, int y) {
  if(x >= 0 && x < width && y >= 0 && y < height) {
    // and it's a floor tile, not a wall tile:
    switch (getChar(x, y)) {
    case 'r': return 1;
    case 'g': return 2;
    case 'b': return 3;
    case 'y': return 4;
    case '#': return 0;
    default: return 0;
    }
  }
  return 0;
}
```

2.5 Filtering

Filtering is to compute the result for $t + 1$ from the new evidence e_{t+1}, given the result of filtering up to time t.

$$P(X_{t+1}|e_{1:t+1}) = f(e_{t+1}, P(X_t|e_{1:t}))$$

The function f is just to represent there is a function f can satisfy the process: recursive estimation.

Skipping all the inferences, according to the textbook, the filtering equation is :

$$P(L_{t+1}|C_{1:t+1}) = \alpha P(C_{t+1}|L_{t+1})P(L_{t+1}|C_{1:t}) \quad (1)$$
$$= \alpha P(C_{t+1}|L_{t+1}) \sum_{l_t} P(L_{t+1}|l_t)P(l_t|C_{1:t}) \quad (2)$$

In my method, I use two steps to compute the $P(L_{t+1}|C_{1:t+1})$.

1.
$$P(L_{t+1}|C_{1:t}) = \sum_{l_t} P(L_{t+1}|l_t)P(l_t|C_{1:t}). \quad (3)$$

2.
$$P(L_{t+1}|C_{1:t+1}) = \alpha P(C_{t+1}|L_{t+1})P(L_{t+1}|C_{1:t}). \quad (4)$$

The α is a normalizing constant used to make probabilities sum up to 1.

In the blind robot problem, we need to get a sequence of probability distributions describing the possible locations of the robot at each time. So I set a `getProbDistr` function to solve the problem in `BlindRobotMazeProblem.java`. The function gets the necessary information about maze from the class and sensor reading sequence from the input of the function. Then it can computes the sequence of probability distributions that we want. Here is my code for `getProbDistr`:

```
1  // get probability distributions describing the possible locations
2  //   of the robot at each time step.
3  public ArrayList<double[][]> getProbDistr(int[] colors) {
4    return Filtering(colors);
5  }
6
7  // Filtering: Forward P(Xt|e1:t)
8  private ArrayList<double[][]> Filtering(int[] colors) {
9    // probability distributions sequence of each steps.
10   ArrayList<double[][]> probDistrSeq = new ArrayList<double[][]>();
11
12   // before first step, initial probability distribution: P(L0)
13   double[][] prob_L0 = new double[maze.height][maze.width];
14
15   // get the num of valid locations except walls
16   int num = 0;
17   for (int i = 0; i < maze.height; i++)
18     for (int j = 0; j < maze.width; j++)
19       if (maze.isLegal(j, i))
20         num++;
21
22   for (int i = 0; i < maze.height; i++)
23     for (int j = 0; j < maze.width; j++)
24       if (maze.isLegal(j, i))
25         prob_L0[i][j] = 1.0 / num;
26
27   probDistrSeq.add(prob_L0);
28
29   // one step, get a color as evidence variable Ct
30   // Then do filtering to get P(Xt|e1:t) = P(Lt|c1:t)
31   for (int color: colors) {
32     // step1: P(Xt+1|e1:t) = sum_xt(P(Xt+1|Xt)*P(Xt|e1:t))
33     //   P(Xt+1|Xt) got from transition model
```

```
34        //   P(Xt|e1:t) is last step probability distribution
35
36        // P(Xt+1|e1:t) = P(Lt+1|c1:t)
37        double[][] prob_t1_t = new double[maze.height][maze.width];
38
39        // P(Xt|e1:t) = P(Lt|c1:t)
40        double[][] prob_t_t = probDistrSeq.get(probDistrSeq.size()-1);
41
42        for (int i = 0; i < maze.height; i++)
43          for (int j = 0; j < maze.width; j++) {
44            // Location (j,i)
45            prob_t1_t[i][j] = 0;
46            for (int m = 0; m < maze.height; m++)
47              for (int n = 0; n < maze.width; n++) {
48                // transition (n,m)->(j,i)
49                ArrayList<Integer> trans = new ArrayList<Integer>();
50                trans.add(n);
51                trans.add(m);
52                trans.add(j);
53                trans.add(i);
54                // in transModel, only store nonzero transition
55                if (transModel.containsKey(trans))
56                  prob_t1_t[i][j] += transModel.get(trans) * prob_t_t[m][n];
57              }
58          }
59
60        // step2: P(Xt+1|e1:t+1) = alpha * P(et+1|Xt+1)*P(Xt+1|e1:t)
61        //   P(et+1|Xt+1) got from sensor model
62        //   P(Xt+1|e1:t) got from step1
63
64        // P(Xt+1|e1:t+1) = P(Lt+1:c1:t+1)
65        double[][] prob_t1_t1 = new double[maze.height][maze.width];
66        double sum = 0;
67
68        // P(et+1|Xt+1)*P(Xt+1|e1:t)
69        for (int i = 0; i < maze.height; i++)
70          for (int j = 0; j < maze.width; j++) {
71            // getP(et+1|Xt+1) from sensor model: Location (j,i)-> Ct+1
72            prob_t1_t1[i][j] = sensorModel(j,i, color) * prob_t1_t[i][j];
73            sum += prob_t1_t1[i][j];
74          }
75        // alpha
76        for (int i = 0; i < maze.height; i++)
77          for (int j = 0; j < maze.width; j++) {
78            prob_t1_t1[i][j] = prob_t1_t1[i][j] / sum;
79          }
80
81        probDistrSeq.add(prob_t1_t1);
82      }
83
84      return probDistrSeq;
85    }
```

Generally, I use ad hoc to compute the probabilities of each legal location in the maze. The probability distribution stored in ArrayList probDistrSeq is type of `double[][]`, which is a 2-dimensional array of double the same as `grid` in provided code `Maze.java`.

1. Initially, I have no observations, only the robot's prior beliefs. So assume that every legal location has a probability (1/ number of legal locations) and the probability of illegal locations, walls, is 0. Line 7-20 is about this step.

2. Get a color in the sequence orderly.

3. Compute equation 3 for each location.

4. Compute equation 4 using the results from last step for each location. Those are probability distributions for this step.

2.6 Exhibit results

I changed the provided codes in assignment 2, mazeworld, to showing the actual motions of the robot and the probability distributions. The results are shown like Figure 2. And the probability distribution will be changed according to the motion of the robot. The white circle shows the actual location of the robot. Also the robot doesn't know where it is but we know. According to the actual location of the robot, we can see whether the probability distribution is reasonable.

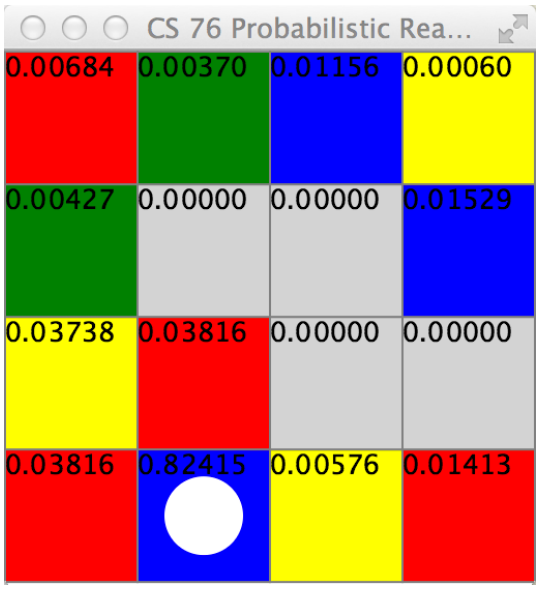

Figure 2: This UI shows probability distributions friendly

I read whether the square is a color or a wall from a data file, and show it in `MazeView.java`. Add some `Text` labels for each square to show the probability of the square. When the robot does an action, the probability distribution will be changed. I call `setText` to change the content in the label to show the current probability. The code is below:

```
1   public MazeView(Maze m, int pixelsPerSquare) {
2     currentColor = 0;
3
4     pieces = new ArrayList<Node>();
5
6     maze = m;
7     this.pixelsPerSquare = pixelsPerSquare;
8
9   //    Color colors[] = { Color.LIGHTGRAY, Color.WHITE };
```

```
10    //  int color_index = 1; // alternating index to select tile color
11
12      labels = new Text[maze.height][maze.width];
13
14      for (int c = 0; c < maze.width; c++) {
15        for (int r = 0; r < maze.height; r++) {
16
17          int x = c * pixelsPerSquare;
18          int y = (maze.height - r - 1) * pixelsPerSquare;
19
20          Rectangle square = new Rectangle(x, y, pixelsPerSquare,
21              pixelsPerSquare);
22
23          square.setStroke(Color.GRAY);
24          switch (maze.getChar(c, r)) {
25          case 'r': square.setFill(Color.RED); break;
26          case 'g': square.setFill(Color.GREEN); break;
27          case 'b': square.setFill(Color.BLUE); break;
28          case 'y': square.setFill(Color.YELLOW); break;
29          default: square.setFill(Color.LIGHTGRAY); break;
30          }
31
32          labels[r][c] = new Text(x, y + 12, "" + c +","+ r);
33
34          this.getChildren().add(square);
35          this.getChildren().add(labels[r][c]);
36        }
37
38      }
39
40    }
41
42    public void setTexts(double[][] probs) {
43      for (int i = 0; i < maze.height; i++)
44        for (int j = 0; j < maze.width; j++) {
45          String str = String.format("%.5f", probs[i][j]);
46          labels[i][j].setText(str);
47        }
48    }
```

The data file of the maze is like below:

```
r g b y
g # # b
y r # #
r b y r
```

`r,g,b,y` represents the color, red, green, blue, yellow. `#` represents a wall. This maze built in the program is the Figure 1.

The locations sequence of the robot is given manually and use the `AnimationPath` in provided code to show it. The `searchNode` type in `AnimationPath` is changed to `int[2]` to represents the location (x,y).

3 Experiments and discussions

I will introduce an 4*4 maze example in the next subsection. I will show the probability distribution of each step in details. And then some examples of larger maze will be shown.

3.1 A 4*4 maze example

The necessary information for the example.

- Initial location: (0,0).
- Action sequence: East, West, West, East.
- Location sequence: (1,0), (0,0), (0,0), (1,0)
- Color sequence: b, r, r, b (sensor gets all right color)

The probability distributions of each step are shown below:

1. The initial probability distribution is in Figure 3.

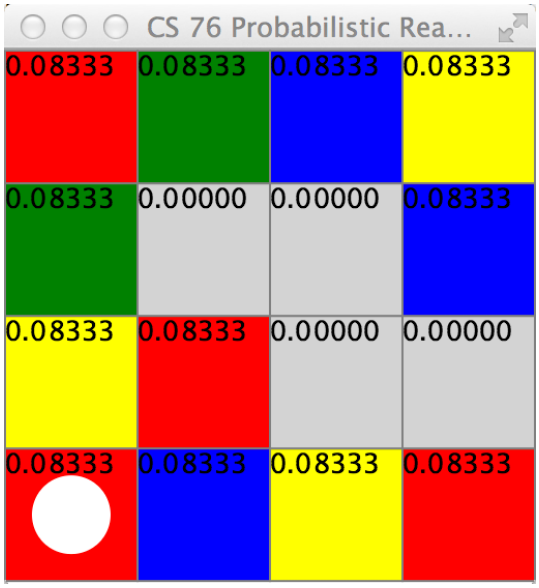

Figure 3: The initial probability distribution of the 4*4 maze

   The legal location has the same probability 0.08333, which is an approximate number. Because the double will show many numbers after the point, for the beauty and clearness, I set the number only can show 5 numbers after the point in the label. The probability of illegal location is 0. The sum of all probability distribution is 1.

2. The 1st step: East. Location: (0,0)->(1,0). Sensor gets color: blue in location (1,0). The probability distribution is in Figure 4.

   Clearly, all blue location have the same probability, 0.29333, while other legal locations has the same probability, 0.01333, because the only evidence is the blue got in this step.

3. The 2nd step: West. Location: (1,0)->(0,0). Sensor gets color: red in location (0,0). The probability distribution is in Figure 5.

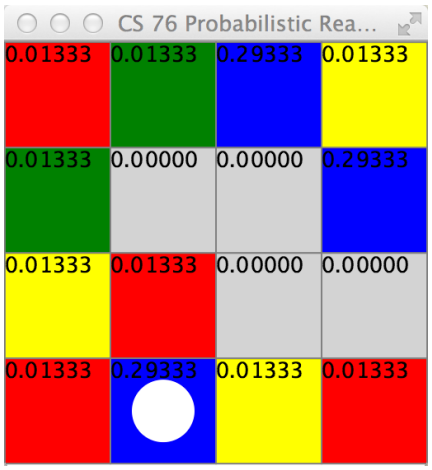

Figure 4: The 1st step probability distribution

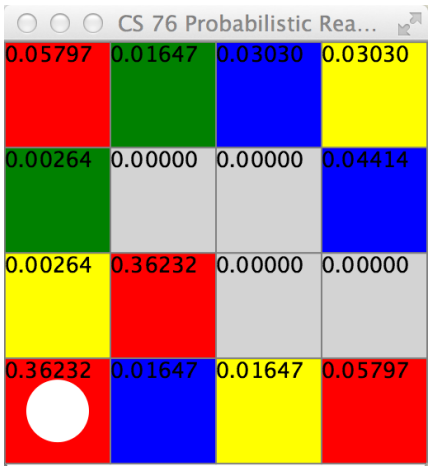

Figure 5: The 2nd step probability distribution

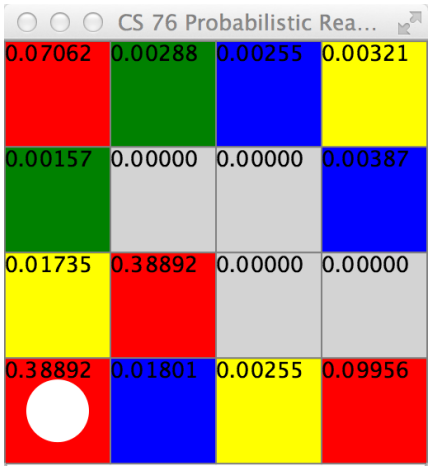

Figure 6: The 3rd step probability distribution

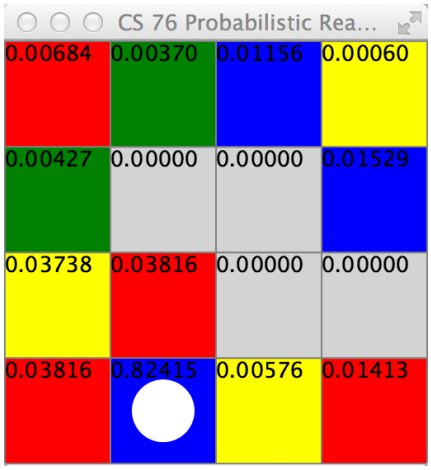

Figure 7: The 4th step probability distribution

In the result, the location (0,0) and (1,1) has the highest probability of all locations. Because the robot doesn't know whether it move or not and which direction the robot heads to, two red location with a blue location next has the same highest probability, 0.36232.

4. The 3rd step: West. Location: (0,0)->(0,0). Sensor gets color: red in location (0,0). The probability distribution is in Figure 6.

   In the result, the location (0,0) and (1,1) has the highest probability of all locations. Only the two location can get the color sequence: blue, red, red, so they has the same highest probability, 0.38892.

5. The 4th step: East. Location: (0,0)->(1,0). Sensor gets color: blue in location (1,0). The probability distribution is in Figure 7.

   In the result, only the location (1,0) has the highest probability of all locations. Because only this location can get the color sequence: blue, red, red, blue. The highest probability is 0.82415.

In this example, finally, only a legal location in the maze can get the sequence of color. Obviously, the location gets the highest probability in the probability distribution. Moreover, the sum of all probabilities is 1, which satisfies the probability theory. And the probability of a wall is always 0.

3.2 A 7*7 maze example

The necessary information for the example.

- Initial location: (0,0).
- Action sequence: East, West, West, East, East, East, North, East, North, East, North, North.
- Location sequence: (1,0), (0,0), (0,0), (1,0), (2,0), (3,0), (3,1), (4,1), (4,2), (5,2), (5,3), (5,4).
- Color sequence: b, r, r, b, y, r, y, b, g, y, y, r (sensor gets all right color).

The probability distributions of each step are shown below:

1. The initial probability distribution is in Figure 8.

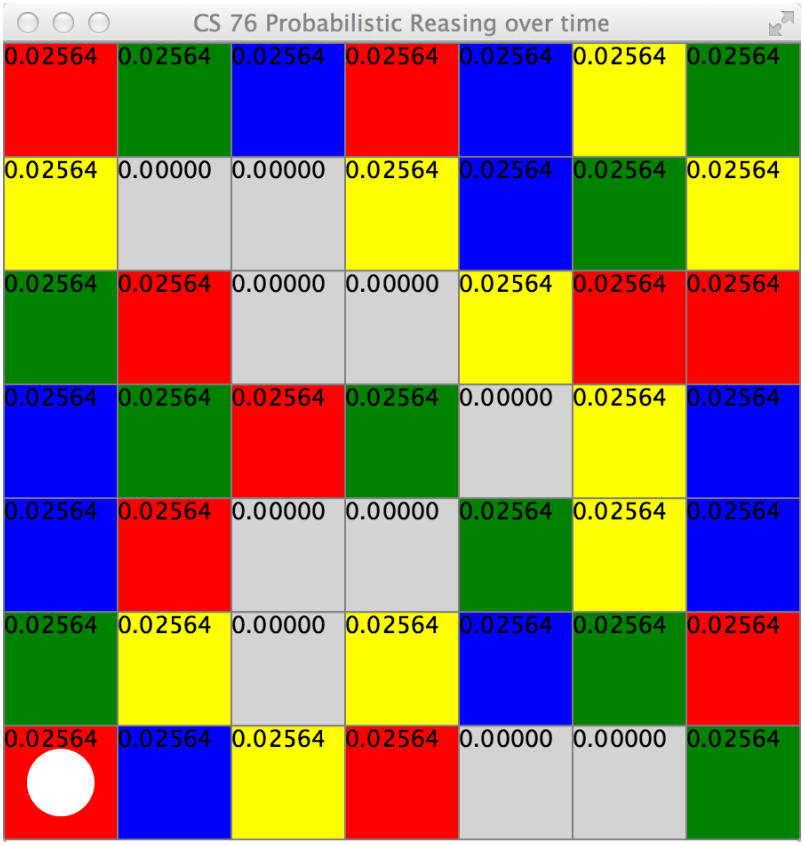

Figure 8: The initial probability distribution of the 7*7 maze

The legal location has the same probability 0.02564. The probability of illegal location is 0. The sum of all probability distribution is 1.

2. The 1st step: East. Location: (0,0)->(1,0). Sensor gets color: blue in location (1,0). The probability distribution is in Figure 9.

   Clearly, all blue location have the same probability, 0.09649, while other legal locations has the same probability, 0.00439, because the only evidence is the blue got in this step.

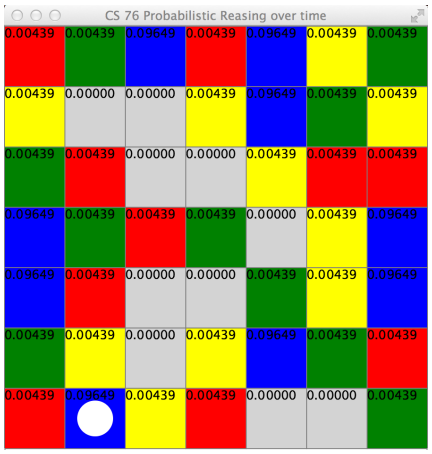

Figure 9: The 1st step probability distribution

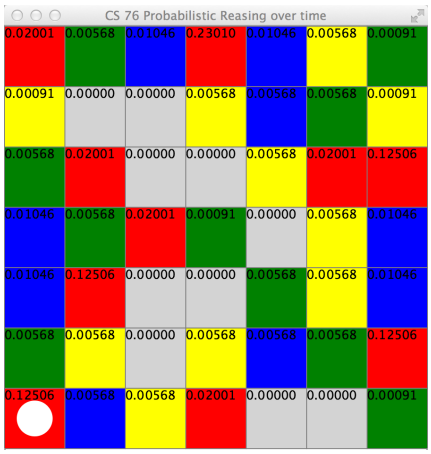

Figure 10: The 2nd step probability distribution

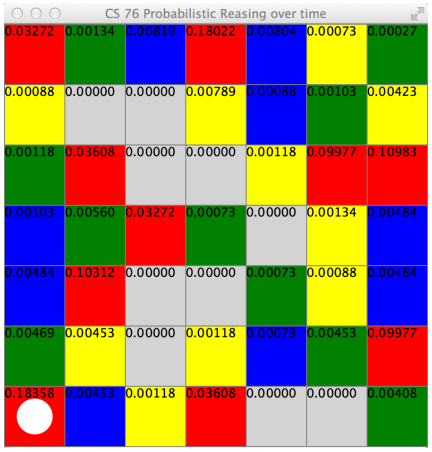

Figure 11: The 3rd step probability distribution

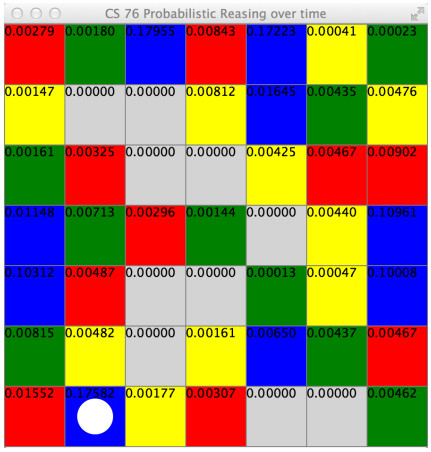

Figure 12: The 4th step probability distribution

3. The 2nd step: West. Location: (1,0)->(0,0). Sensor gets color: red in location (0,0). The probability distribution is in Figure 10.

   In the result, the location (3, 6) has the highest probability 0.23010 of all locations, the actual location (0,0) has the second highest probability 0.12506. Because the red location (3,6) has 2 blue square next to it, it is more likely to move from a blue square, while location (0,0) only has one blue square next to it. There are only two actions be done. Due to this, I can infer that two least actions cannot help the robot find where it is. The robot needs more actions to get a right, more accurate location.

4. The 3rd step: West. Location: (0,0)->(0,0). Sensor gets color: red in location (0,0). The probability distribution is in Figure 11.

   In the result, the location (0,0) has the highest probability 0.18358 of all locations now. The probability distribution this step used get from last step is too various. And the color sequence leads to this result. This step is red->red. (3,6) with the highest probability in last step only has 0.25 chance to get this transition. But (0,0) has 0.5 chance to get this transition. This is the most important element affecting the probabilities.

5. The 4th step: East. Location: (0,0)->(1,0). Sensor gets color: blue in location (1,0). The probability

distribution is in Figure 12.

In the result, only the location (2,6) has the highest probability 0.17955 of all locations, while actual location (1,0) has the second highest probability 0.17582. Although I get the actual location with the highest probability last step, this step gets the wrong location with the highest probability. From this, it predicts that I still need more actions to be close to the right answer. Furthermore, the probability of each location is too small, less than 0.5. It's not enough to proof one location is where the robot is.

6. Let's skip some steps and forward to the last probability distribution. The probability distribution is in Figure 13.

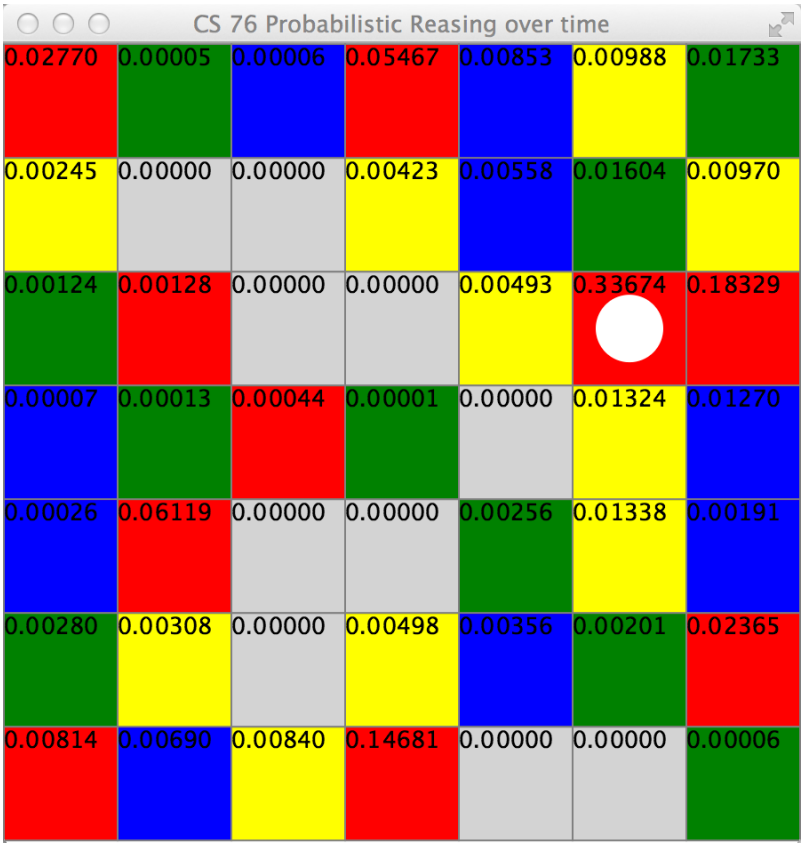

Figure 13: The final probability distribution of the 7*7 maze

In this step, the actual location has the highest probability 0.33674, which is much greater than the probabilities of other locations.

From these two examples above, we can conclude that more actions can help to get the probability more reasonable. When you get a location with highest probability wich may not be the actual location where the robot is, you need to try more actions until the probability is much greater than others. The larger the maze is, the more various the colors in the maze is. So the probability distribution will be more various and change variously. More actions are required to figure out where on earth the robot is.

3.3 A 4*4 maze example with a wrong sensor value

The necessary information for the example.

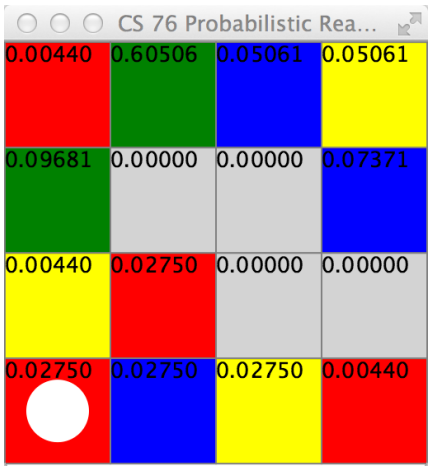

Figure 14: The 2nd step probability distribution

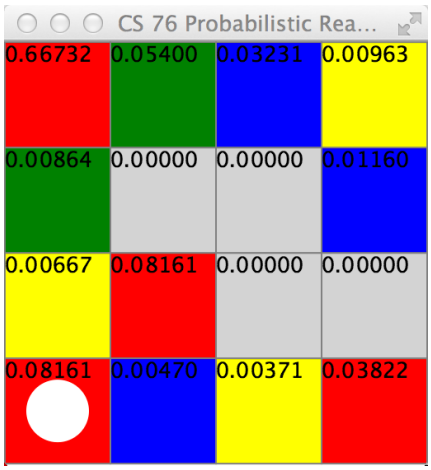

Figure 15: The erd step probability distribution

- Initial location: (0,0).
- Action sequence: East, West, West, East, North, West.
- Location sequence: (1,0), (0,0), (0,0), (1,0), (1,1), (0,1)
- Color sequence: b, g, r, b, r, y (the second color is wrong, the color of the location is red)

The probability distributions of each step are shown below:

1. The initial and the 1st probability distribution is the same with that in the first example, in Figure 3 and Figure 4.

2. The 2nd step: West. Location: (1,0)->(0,0). Sensor gets color: green in location (0,0), which is wrong. The probability distribution is in Figure 14.

   In the result, the location (1,3) has the highest probability 0.60506 of all locations. Because the robot believes that it is in a green location with a blue neighbor now. Because the wrong sensor value, the robot's belief state is far away from the right one.

3. The 3rd step: West. Location: (0,0)->(0,0). Sensor gets color: red in location (0,0). The probability distribution is in Figure 15.

   In the result, the location (0,3) has the highest probability 0.66732 of all locations. Because the robot believes that it is ina red location which is from a green square and the green square has a blue neighbor.

4. The 4th step: East. Location: (0,0)->(1,0). Sensor gets color: blue in location (1,0). The probability distribution is in Figure 16.

   In the result, only the location (1,0) has the highest probability 0.33622 of all locations. Because only the location with the highest probability doesn't has a blue neighbor. Then the robot is correcting the wrong sensor value now. So the actual location (1,0) has the highest probability now.

5. The 5th step: North. Location: (1,0)->(1,1). Sensor gets color: red. The 6th step: East. Location: (1,1)->(0,1). Sensor gets color: yellow in location (0,1). The final probability distribution is in Figure 17.

   In the result, only the location (0,1) has the highest probability 0.69019 of all locations. Because going through all this sensors, the robot finally gets a location with probability much high than others. This is the right location.

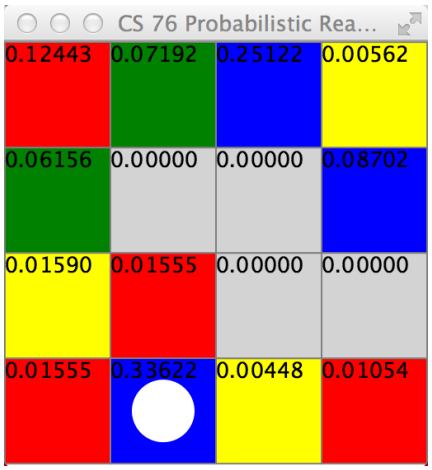

Figure 16: The 4th step probability distribution

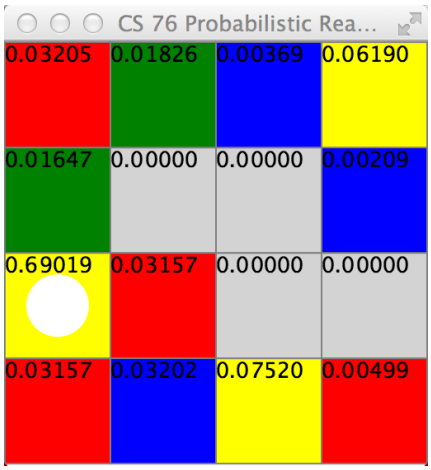

Figure 17: The 6th step probability distribution

From the example of right color sequence and wrong color sequence, I conclude that although there is wrong sensor values in a color sequence, the robot can correct the mistake to get a right location with more actions. Moreover, this probabilistic reasoning over time approach is much stronger than the method I implemented in assignment 2. Because in the blind robot problem in assignemt 2, if there is a wrong sensor value, the robot will throw away the location that doesnot satisfy the condition including the actual location. With probability, the actual location can increase its probability in the next steps, although a mistake may decrease its probability much.

4 Forward-backward smoothing(Bonus)

Smoothing is the process of computing the probability distribution over past states given evidence up to the present. That means to compute $P(X_k|e_{1:t}), 0 \leq k < t$. For smoothing, I will introduce a Forward-backward algorithm here.

The basic idea of forward-backward algorithm is to compute

$$P(X_k|e_{1:t}) = \alpha P(X_k|e_{1:k})P(e_{k+1:t}|X_k)$$

The item $P(X_k|e_{1:k})$ is got from filtering, the Forward algorithm. The next item is something we need to compute here.

$$P(e_{k+1:t}|X_k) = \sum_{x_{k+1}} P(e_{k+1}|x_{k+1})P(e_{k+2:t}|x_{k+1})P(x_{k+1}|X_k)$$

The first item in the right side is from sensor model and the last item is from transition model. The middle one is one step further of $P(e_{k+1:t}|X_k)$. So it is a recursive call.

4.1 Implementation

I implemented the Forward-Backward algorithm as a function `ForwardBackward` in `BlindRobotMazeProblem.java`. Here is the codes:

```
1  // forward-backward algorithm for smoothing P(Xk|e1:t) 0 < k < t
2  private ArrayList<double[][]> ForwardBackward(int[] colors) {
3    // fv[t] = P(Xt|e1:t)
```

```
    ArrayList<double[][]> fv = Filtering(colors);
    // sv[t] =  PXk |e1:t)
    ArrayList<double[][]> sv = new ArrayList<double[][]>();
    // b = P(ek+1:t|Xk)
    double[][] b = new double[maze.height][maze.width];

    // the first b = P(ek+1:t|Xk) = 1
    for (int m = 0; m < maze.height; m++)
      for (int n = 0; n < maze.width; n++) {
        b[m][n] = 1;
      }

    for (int i = fv.size()-1; i > 0; i--) {
      int color = colors[i-1];
      //sv[i] = Normalize(fv[i] x b)
      //sv[i] = alpha P(Xk|e1:k) * P(ek+1:k|Xk)
      double[][] svi = new double[maze.height][maze.width];
      double[][] fvi = fv.get(i);

      // P(Xk|e1:k) * P(ek+1:t|Xk)
      double sum = 0;
      for (int m = 0; m < maze.height; m++)
        for (int n = 0; n < maze.width; n++) {
          svi[m][n] = fvi[m][n] * b[m][n];
          sum += svi[m][n];
        }

      // alpha
      for (int m = 0; m < maze.height; m++)
        for (int n = 0; n < maze.width; n++) {
          svi[m][n] = svi[m][n] / sum;
        }

      sv.add(0, svi);

      // b = Backward(b,ev[i])
      // b = P(ek:t|Xk-1) = sum_xk(P(ek|xk) * P(ek+1:t|xk) * P(xk|xk-1))
      // Xk-1 = (n,m)
      double[][] tmp = new double[maze.height][maze.width];

      for (int m = 0; m < maze.height; m++)
        for (int n = 0; n < maze.width; n++) {
          // xk = (l,k)
          for (int k = 0; k < maze.height; k++)
            for (int l = 0; l < maze.width; l++) {
              // P(xk|xk-1)
              ArrayList<Integer> trans = new ArrayList<Integer>();
              trans.add(m);
              trans.add(n);
              trans.add(k);
              trans.add(l);
              if (transModel.containsKey(trans))
                tmp[m][n] += sensorModel(l,k, color) * b[k][l] * transModel.get(trans);
            }
        }
      b = tmp;
```

```
60      }
61
62      // P(X0|e1:t) = alpha P(X0) * P(e1:t|X0) = alpha * P(X0) * b
63      double[][] px0 = new double[maze.height][maze.width];
64      double sum = 0;
65      for (int m = 0; m < maze.height; m++)
66        for (int n = 0; n < maze.width; n++) {
67          px0[m][n] = fv.get(0)[m][n] * b[m][n];
68          sum += px0[m][n];
69        }
70
71      for (int m = 0; m < maze.height; m++)
72        for (int n = 0; n < maze.width; n++) {
73          px0[m][n] /= sum;
74        }
75      sv.add(0, px0);
76
77      return sv;
78
79    }
```

In this function,

1. Compute the $fv[i] = P(X_t|e_{1:t})$ by forward algorithm first. In my method, I call `Filtering` in line 4.
2. Then do Backward algorithm. Compute the $sv[i] = P(X_k|e_{1:t})$ from the last step to the first step.
   sv[i] = NORMALIZE(fv[i-1] x b):

$$P(X_k|e_{1:t}) = \alpha P(X_k|e_{1:k})P(e_{k+1:t}|X_k)$$

   b = BACKWARD(b, ev[i]):

$$P(e_{k+1:t}|X_k) = \sum_{x_{k+1}} P(e_{k+1}|x_{k+1})P(e_{k+2:t}|x_{k+1})P(x_{k+1}|X_k)$$

3. In my method, set the initial $b = P(e_{t+1}|X_t) = 1$ for each X_t.
4. In my method, after computing $P(X_k|e_{1:t})$ for each $X_1, X_2, ..., X_t$, compute $P(X_0|e_{1:t})$ using eqution:

$$\begin{aligned} P(X_0|e_{1:t}) &= alphaP(X_0)P(e_{1:t}|X_0) &(5)\\ &= alphaP(X_0) * b &(6)\end{aligned}$$

4.2 Experiment and Discussion

Using the 4*4 maze example in section 3.1 to exhibit the results. I will exhibit the probability distribution for each step in the squares.

The necessary information for the example.

- Initial location: (0,0).
- Action sequence: East, West, West, East.
- Location sequence: (1,0), (0,0), (0,0), (1,0)

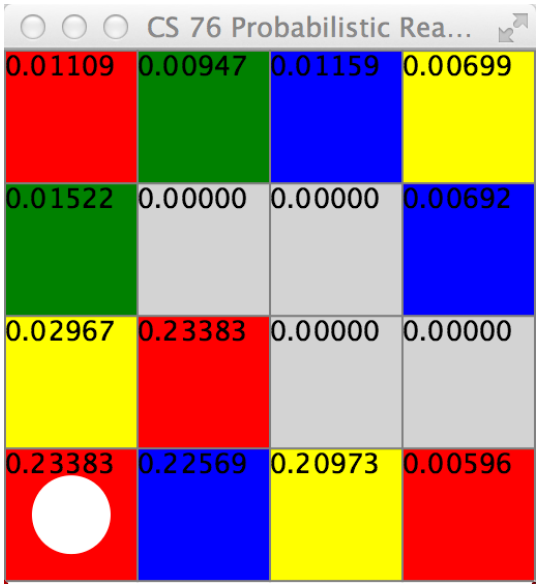

Figure 18: The initial probability distribution of the 4*4 maze

- Color sequence: b, r, r, b (sensor gets all right color)

The probability distributions of each step are shown below:

1. The initial probability distribution is in Figure 18.

   The location (0,0) and (1,1) have the same probability 0.23383, which is more accurate than the result in Forwad algorithm in last section. Because Forward-backward considers the evidence after this step, it can use the $C_1 = b, C_2 = r$ to find (0,0) and (1,1) are better than other legal locations.

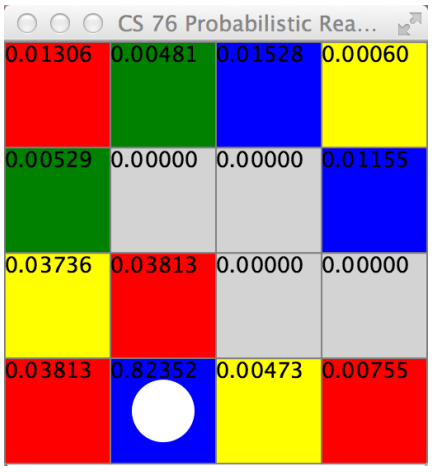

Figure 19: The 1st step probability distribution

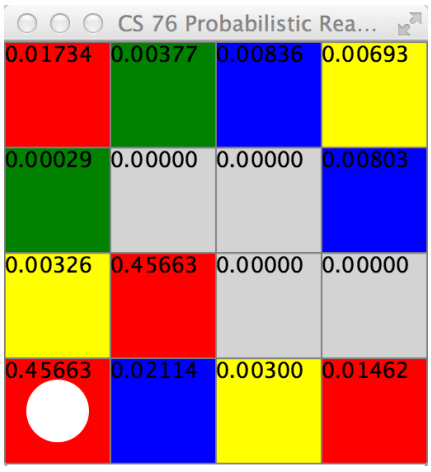

Figure 20: The 2nd step probability distribution

2. The 1st step: East. Location: (0,0)->(1,0). Sensor gets color: blue in location (1,0). The probability distribution is in Figure 19.

   The location with highest probability is (1,0). Obviously, the probability distribution is better than

forward algorithm, which shows all blue location have the same probability. Because forward-backward takes care of evidence in advance.

3. The 2nd step: West. Location: (1,0)->(0,0). Sensor gets color: red in location (0,0). The probability distribution is in Figure 20.

   In the result, the location (0,0) and (1,1) has the highest probability of all locations. Because the robot doesn't know whether it move or not and which direction the robot heads to, two red location with a blue location next has the same highest probability, 0.45663.

4. The 3rd step: West. Location: (0,0)->(0,0). Sensor gets color: red in location (0,0). The probability distribution is in Figure 21.

   In the result, the location (0,0) and (1,1) has the highest probability of all locations. Only the two location can get the color sequence: blue, red, red, so they has the same highest probability, 0.38892. I found that the probability is less than the probability we got in last step. Because we compute the probability by backward.

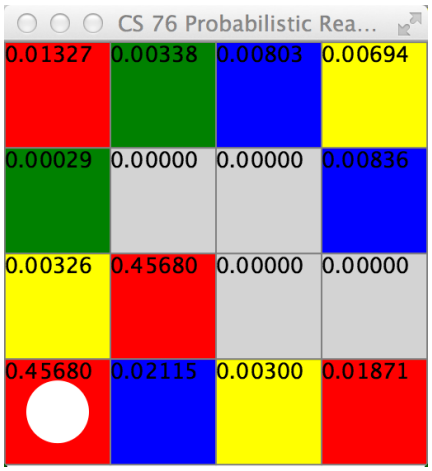

Figure 21: The 3rd step probability distribution

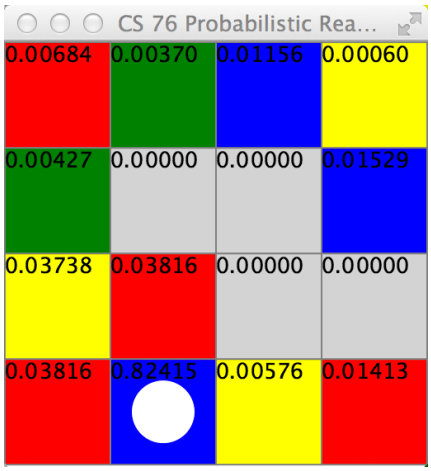

Figure 22: The 4th step probability distribution

5. The 4th step: East. Location: (0,0)->(1,0). Sensor gets color: blue in location (1,0). The probability distribution is in Figure 22.

   In the result, only the location (1,0) has the highest probability of all locations, 0.82415. This probability distribution is the same with that in foward algorithm. Because $b = 1$ in this step.

In this example, we can found that the probability distributions are better than those in forward algorithm in last section. Obiously, forward-backward is better than only forward. We can exclude more unreasonable locations with lower probability.

5 Compute the most likely sequence of robot locations(Bonus)

Above, I talked about how to get the probability distribution of belief states. After getting these probability distributions, we can find the most likely explanations. In this blind robot problem, we can compute the most likely sequence of robot locations. Here comes a introduction of Viterbi algorithm.

The *Viterbi algorithm* is a dynamic programming algorithm to find the most likely sequence of hidden states.

The basic idea is to compute the equation below to get the most likely sequence of $\{x_1, ..., x_t, x_{t+1}\}$.

$$\max_{x_1 \ldots x_t} P(x_1, ..., x_t, X_{t+1}|e_{1:t+1})$$
$$= \alpha P(e_{t+1}|X_{t+1}) \max_{X_t} \left(P(X_{t+1}) \max_{x_1 \ldots x_{t-1}} P(x_1, ..., x_{t-1}, x_t|e_{1:t}) \right) \quad (7)$$

For each value of X_{t+1}, get

$$\max_{x_1 \ldots x_t} P(x_1, ..., x_t, X_{t+1}|e_{1:t+1})$$

And then, find the maximum value of X_{t+1}, the sequence of $\{x_1, ..., x_t, x_{t+1}\}$ is the most likely sequence. In this specific problem, it is the most likely sequence of robot locations.

5.1 Implementation

I implemented the Viterbi algorithm as a function `getPath_Viterbi` in `BlindRobotMazeProblem.java`. Here is the codes:

```
1   // using Viterbi to get the most likely sequence, maybe more than one
2   public ArrayList<ArrayList<int[]>> getPath_Viterbi(int[] colors, ArrayList<double[][]>
        probDistr) {
3
4     // P(x1,...,xt|e1:t)
5     // Initial probability distribution: P(L0)
6     double[][] prob_xt = new double[maze.height][maze.width];
7
8     // get the num of valid locations except walls
9     int num = 0;
10    for (int i = 0; i < maze.height; i++)
11      for (int j = 0; j < maze.width; j++)
12        if (maze.isLegal(j, i))
13          num++;
14
15    for (int i = 0; i < maze.height; i++)
16      for (int j = 0; j < maze.width; j++)
17        if (maze.isLegal(j, i))
18          prob_xt[i][j] = 1.0 / num;
19
20    probDistr.add(prob_xt);
21
22    // keep maxarg xt
23    // ArrayList[0] = null, because no location before L0
24    // key: L[x,y] value: locations(x2,y2) lead to this (x,y)
25    ArrayList<HashMap<ArrayList<Integer>, ArrayList<int[]>>> maxarg_xt_list = new
          ArrayList<HashMap<ArrayList<Integer>, ArrayList<int[]>>>();
26    maxarg_xt_list.add(null);
27
28    // max P(x1,...xt, Xt+1| e1:t+1) = alpha P(et+1|Xt+1) max( P(Xt+1|xt) max(x1,...,xt|e1:t) )
29    for (int color: colors) {
30      // max P(x1,...xt, Xt+1| e1:t+1)
31      double[][] prob = new double[maze.height][maze.width];
32
33      // keep maxarg xt in this step
34      HashMap<ArrayList<Integer>, ArrayList<int[]>> maxarg_xt = new HashMap<ArrayList<Integer>,
            ArrayList<int[]>>();
35
```

```
36        // for each Xt+1, prob[i][j] = max( P(Xt+1|xt) max(x1,...,xt|e1:t) )
37        for (int i = 0; i < maze.height; i++)
38          for (int j = 0; j < maze.width; j++) {
39
40            // keep max
41            double max = 0;
42            ArrayList<int[]> maxLoc = new ArrayList<int[]>();
43
44            // for each xt, max ( P(Xt+1|xt) max(x1,...,xt|e1:t) )
45            for (int m = 0; m < maze.height; m++)
46              for (int n = 0; n < maze.width; n++) {
47                ArrayList<Integer> trans = new ArrayList<Integer>();
48                trans.add(m);
49                trans.add(n);
50                trans.add(i);
51                trans.add(j);
52                if (transModel.containsKey(trans)) {
53                  double tmp = transModel.get(trans) * prob_xt[m][n];
54                  if (tmp > max) {
55                    max = tmp;
56                    if (!maxLoc.isEmpty())
57                      maxLoc.clear();
58                    maxLoc.add(new int[]{n,m});
59                  }
60                  else if (tmp == max) {
61                    maxLoc.add(new int[]{n,m});
62                  }
63                }
64              }
65            prob[i][j] = max;
66            ArrayList<Integer> loc = new ArrayList<Integer>();
67            loc.add(j);
68            loc.add(i);
69            maxarg_xt.put(loc, maxLoc);
70          }
71
72        // P(et+1|Xt+1) max( P(Xt+1|xt) max(x1,...,xt|e1:t) )
73        double sum = 0;
74        for (int i = 0; i < maze.height; i++)
75          for (int j = 0; j < maze.width; j++) {
76            prob[i][j] = sensorModel(j,i, color) * prob[i][j];
77            sum += prob[i][j];
78          }
79
80        // alpha
81        for (int i = 0; i < maze.height; i++)
82          for (int j = 0; j < maze.width; j++) {
83            prob[i][j] = prob[i][j]/sum;
84          }
85
86        // prepare for next loop
87        prob_xt = prob;
88        probDistr.add(prob_xt);
89        maxarg_xt_list.add(maxarg_xt);
90
91      }
```

```
92
93      // maxarg P(x1,...xt, Xt+1| e1:t+1) find the path
94      // find maxarg P(x1,...xt, Xt+1| e1:t+1)
95      double max = 0;
96      int[] maxLoc = new int[2];
97      for (int i = 0; i < maze.height; i++)
98        for (int j = 0; j < maze.width; j++) {
99          if (prob_xt[i][j] > max) {
100           max = prob_xt[i][j];
101           maxLoc[0] = j;
102           maxLoc[1] = i;
103         }
104       }
105
106     // find path
107     ArrayList<ArrayList<int[]>> paths = new ArrayList<ArrayList<int[]>>();
108     ArrayList<int[]> p = new ArrayList<int[]>();
109     p.add(0, maxLoc);
110
111     findPath(maxarg_xt_list, p, paths);
112
113     return paths;
114   }
115
116   private void findPath(ArrayList<HashMap<ArrayList<Integer>, ArrayList<int[]>>> maxarg_xt_list,
117       ArrayList<int[]> p, ArrayList<ArrayList<int[]>> paths) {
118     // if path is complete
119     if (maxarg_xt_list.size() == p.size()) {
120       paths.add(p);
121       return;
122     }
123
124     // find 1 step previous locations
125     HashMap<ArrayList<Integer>, ArrayList<int[]>> maxarg_xt =
          maxarg_xt_list.get(maxarg_xt_list.size()-p.size());
126     ArrayList<Integer> loc = new ArrayList<Integer>();
127     loc.add(p.get(0)[0]);
128     loc.add(p.get(0)[1]);
129     ArrayList<int[]> locs = maxarg_xt.get(loc);
130
131     for (int j = 1; j < locs.size(); j++) {
132       ArrayList<int[]> np = new ArrayList<int[]>(p);
133       np.add(0, locs.get(j));
134       findPath(maxarg_xt_list, np, paths);
135     }
136     p.add(0, locs.get(0));
137     findPath(maxarg_xt_list, p, paths);
138   }
```

In this function. There are 2 steps:

1. Using Viterbi algorithm to compute the equation 7.
2. Find the sequence of $\{x_1, ..., x_t, x_{t+1}\}$ to get the maximum result in last step. The step of finding path is implemented in `findPath` in above codes.

5.2 Experiment and Discussion

Using the 4*4 maze example in section 3.1 to exhibit the results. I exhibit the probability distribution for each step in the squares for convenience that you can see it is reasonable to choose this location in the most likely sequence.

The description of example:

- Initial location: (0,0).
- Action sequence: East, West, West, East.
- Location sequence: (1,0), (0,0), (0,0), (1,0).
- Color sequence: b, r, r, b (sensor gets all right colors)

In this example, there are more than one most likely sequence with the same probability. The sequences are in Figure 23.

```
(1,0) (1,0) (1,1) (1,1) (1,0)
(2,0) (1,0) (1,1) (1,1) (1,0)
(1,1) (1,0) (1,1) (1,1) (1,0)
(0,0) (1,0) (1,1) (1,1) (1,0)
(1,0) (1,0) (0,0) (0,0) (1,0)
(2,0) (1,0) (0,0) (0,0) (1,0)
(1,1) (1,0) (0,0) (0,0) (1,0)
(0,0) (1,0) (0,0) (0,0) (1,0)
```

Figure 23: Most likely paths in a 4*4 maze

1. The initial probability distribution is in Figure 24.

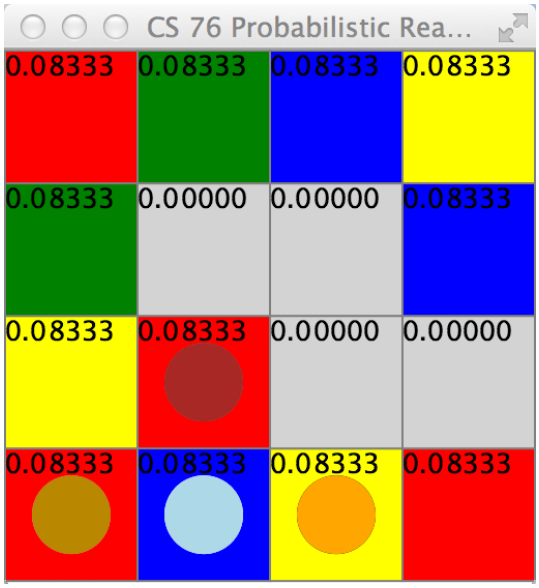

Figure 24: The initial probability distribution of the 4*4 maze and paths

The legal location has the same probability 0.08333. There are actually 8 robots in the maze. You can see the 8 paths in the outputs in Figure 23. The circle represents robots are overlap in the maze.

2. The 1st step: East. Location: (0,0)->(1,0). Sensor gets color: blue in location (1,0). The probability distribution is in Figure 25.

   All robots in different most likely paths are in location (1,0). Although the probability of this location in this step is not very high, the viterbi chooses it finally. According to the actual location we know, but robot doesn't know, viterbi gets the right choice.

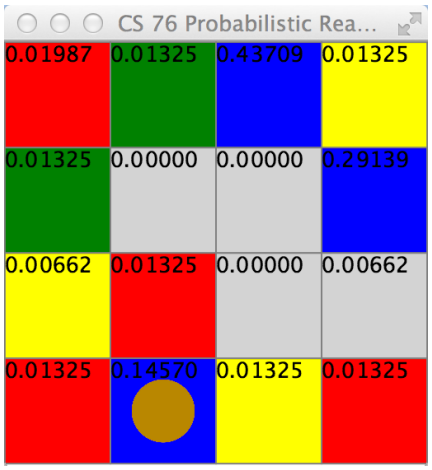

Figure 25: The 1st step probability distribution and paths

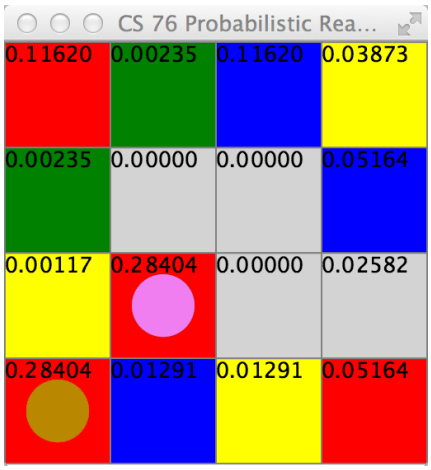

Figure 26: The 2nd step probability distribution and paths

3. The 2nd step: West. Location: (1,0)->(0,0). Sensor gets color: red in location (0,0). The probability distribution is in Figure 26.

   The robots in 8 paths in this step are in (0,0) and (1,1), which have the same probability.

4. The 3rd step: West. Location: (0,0)->(0,0). Sensor gets color: red in location (0,0). The probability distribution is in Figure 27.

   The robots in 8 paths in this step are still in (0,0) and (1,1), which have the same probability.

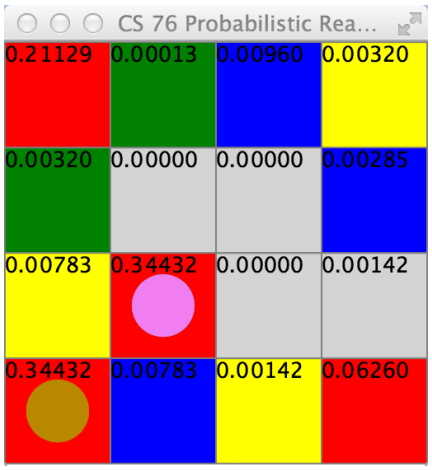

Figure 27: The 3rd step probability distribution and paths

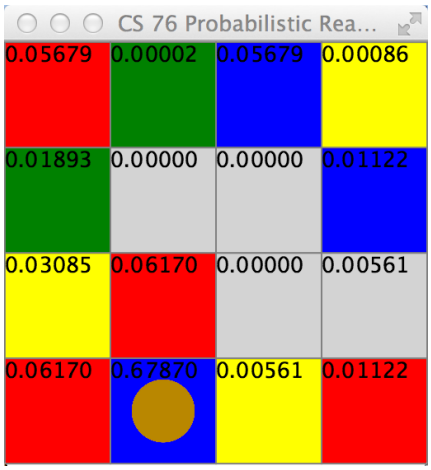

Figure 28: The 4th step probability distribution and paths

5. The 4th step: East. Location: (0,0)->(1,0). Sensor gets color: blue in location (1,0). The probability distribution is in Figure 28.

   All robots in this step are in (1,0) with the highest probability of all locations. In the viterbi, we choose the maximum probability of all locations in this probability distribution. So (1,0) is chosen, then we go back to find the path. That's why we find 8 paths and the location sequences are above.

   In this example, viterbi find the actual path in one of 8 most likely paths. But it cannot certain which one in the 8 paths is better. By adding more actions, the robot can get the most likely path more accurate.

   If use a larger maze as an example, the maze is more complicated and the most likely path that viterbi gets probably is far away from the actual path. I use the 7*7 example in Section 3.2 as an example. The most likely paths shown in Figure 29 are too different from the actual path and no including the actual path, which is (1,0), (0,0), (0,0), (1,0), (2,0), (3,0), (3,1), (4,1), (4,2), (5,2), (5,3), (5,4)!! That is because of the variety of the maze, the most likely sequences we get has a higher probability than the actual one. More actions added, the results can be better.

```
(5,3) (6,3) (6,4) (6,4) (6,3) (5,3) (5,4) (4,4) (4,5) (5,5) (6,5) (6,5) (6,4)
(6,3) (6,3) (6,4) (6,4) (6,3) (5,3) (5,4) (4,4) (4,5) (5,5) (6,5) (6,5) (6,4)
(6,4) (6,3) (6,4) (6,4) (6,3) (5,3) (5,4) (4,4) (4,5) (5,5) (6,5) (6,5) (6,4)
(6,2) (6,3) (6,4) (6,4) (6,3) (5,3) (5,4) (4,4) (4,5) (5,5) (6,5) (6,5) (6,4)
```

Figure 29: Most likely paths in a 7*7 maze