Probabilistic Reasoning Over Time

Robin Wang

March 9, 2014

1 Introduction

This report focuses on a problem based on Rob Schapire, a professor at Princeton. We are given a 4x4 maze with colored cells and a robot that does not know its location or the direction it's moving in, and we need to figure out the location that the robot is most likely at after each move. The robot does not know either its initial location or how it moves, although it does know that it only makes one move per time step. Since the robot does not know its initial location nor the direction of each of its movements, we can only rely on the robot's sensors to estimate the most probable location it is at. Each cell in the maze is one of four colors: red, green, blue, and yellow. The robot's sensor is able to return the correct color of the cell 88% of the time and one of the three wrong colors the rest of the time. The task for this problem now is to determine the probability distribution describing the possible locations of the robot at each time step.

2 Filtering

One method to determine the probability distribution is to use filtering. Filtering, also called state estimation, completes the task of computing the belief state-the posterior distribution over the most recent state?given all evidence, or sensor data, to date. With filtering, we are attempting to calculate $P(X_t|e_{1:t})$, which in other words is the probability of the robot being at some location X_t given the sequence of sensor readings of colors up to time t.

To implement filtering, we would need to understand both the transition model as well as the sensor model. The transition model specifies the probability distribution over the latest state variables given the previous values, $P(X_t|X_{0:t\text{-}1})$. By making a Markov assumption that the current state depends on only a finite fixed number of previous states, we can equate $P(X_t|X_{0:t\text{-}1})$ to $P(X_t|X_{t\text{-}1})$. In terms of our problem, at time t, this is simply the probability of arriving at location X_t given that the robot was just at location $X_{t\text{-}1}$. The sensor model deals with the evidence, or sensor readings. Our sensor model determines $P(E_t|X_t)$, which is the probability of reading in the color at time t given that the robot was at location X_t. To implement our filtering method, given that there are several possible values for the state variable, we will have to keep track of all of the state variables in a vector. So essentially we will have work with a single location X and work with both the transition model and sensor model for X for all locations in the maze. This becomes more clear in our implementation.

2.1 Implementation

A useful filtering algorithm needs to maintain a current state estimate and update it, rather than going back over the entire history of percepts for each update, so 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}. So, to determine $P(X_{t+1}|e_{1:t+1})$, which is $\alpha P(e_{t+1}|X_{t+1})P(X_{t+1}|e_{1:t})$ after dividing up the evidence and using Baye's rule and the sensor's Markov assumption.

To implement this problem we reused much of the code from the `Mazeworld` assignment. I modified several components of `MazeView` and `Maze` in order to display colored cells on the maze and to read these

colored cells based on the `simple.maz` file. In particular, I modified `MazeView`'s constructor so that it reads r, g, b, and y as valid cells and colors them red, green, blue, and yellow respectively. I then modified `Maze`'s `isLegal` method to allow for these changes as well as adding in `isr`, `isg`, `isb`, `isy` to determine if the input cell is a certain color. Aside from these minor modifications, nothing else was done. The majority of the actual implementation of the filtering algorithm is within the `SimpleMazeDriver`.

There were several class variables we declared/initiated in the `SimpleMazeDriver`:

```
// Filtering method
// Path containing the colors of the robot path
ArrayList<String> colorPath = new ArrayList<String>();
// Probability distribution of the probable robot locations
ArrayList<Double> filterDist = new ArrayList<Double>();
// Contains the integer representation and coordinates of each cell
public HashMap<Integer, List<Integer>> intcor;
// Vice - versa
public HashMap<List<Integer>, Integer> intrep;
// For forward-backward smoothing implementation
// Probability distribution
ArrayList<Double> fbDist = new ArrayList<Double>();
// Contains the probability distribution at any time t
public HashMap<Integer, ArrayList<Double>> forwards = new HashMap<Integer, ArrayList<Double>>();
// For Viterbi implementation
// Probability distribution
public ArrayList<Double> ViterbiProbs = new ArrayList<Double>();
// Hashmaps to help store best paths
public HashMap<Integer, List<Integer>> tempbestPath = new HashMap<Integer, List<Integer>>();
public HashMap<Integer, List<Integer>> bestPath = new HashMap<Integer, List<Integer>>();
```

We then modify the `startMoves` method to initialize the robot's path:

```
private void startMoves() {
    // Maze width and height
    int n = maze.width;
    int m = maze.height;
    // Initial probabilities are all 1/16
    for (int i = 0; i < n * m; i++) {
        filterDist.add((double) 1 / 16);
        ViterbiProbs.add((double) 1 / 16);
    }
    // Map domain integers to their coordinates to aid in printing solution
    intcor = new HashMap<Integer, List<Integer>>();
    intrep = new HashMap<List<Integer>, Integer>();
    int key = -1;
    for (int row = 0; row < m; row++) {
        for (int col = 0; col < n; col++) {
            key++;
            intcor.put(key, Arrays.asList(row, col));
            intrep.put(Arrays.asList(row, col), key);
            bestPath.put(key, Arrays.asList(key));
            tempbestPath.put(key, new ArrayList<Integer>());
        }
    }
```

```
        // X and Y coordinates
        List<Integer> x = new ArrayList<Integer>();
        List<Integer> y = new ArrayList<Integer>();
        // Adding in the coordinates of the path of the robot
        x.add(0);
        y.add(0);
        // Add in as many coordinates as desired
        // Path; animation
        animationPathList = new ArrayList<AnimationPath>();
        animationPathList.add(new AnimationPath(mazeView, x, y));
    }
```

We then adjust the inner class `AnimationPath` to animate the robot and call upon our filtering algorithm at each time slice:

```
private class AnimationPath {
        // Robot
        private Node piece;
        // X and Y coordinates
        private List<Integer> xsearchPath;
        private List<Integer> ysearchPath;
        // Time
        private int currentMove = 0;
        // Keep track of location
        private int lastX;
        private int lastY;
        // Help with animation
        boolean animationDone = true;
        // To find out when the entire color sequence is finished
        boolean fb = false;
        // Constructor
        public AnimationPath(MazeView mazeView, List<Integer> x, List<Integer> y) {
            // Get the x,y coordinates
            xsearchPath = x;
            ysearchPath = y;
            // Put the robot down on its first location
            piece = mazeView.addPiece(x.get(0), y.get(0));
            lastX = x.get(0);
            lastY = y.get(0);
            currentMove = 1;
            // Get color
            String color = getColor(lastX, lastY);
            colorPath.add(color);
            // Call on the first step of the filtering algorithm
            predict(color, filterDist);
        }
        // Next time slice
        public void doNextMove() {
            // Check for animation finish
            if (currentMove < xsearchPath.size() && animationDone) {
                // Move the robot
                int dx = xsearchPath.get(currentMove) - lastX;
```

```
            int dy = ysearchPath.get(currentMove) - lastY;
            animateMove(piece, dx, dy);
            lastX = xsearchPath.get(currentMove);
            lastY = ysearchPath.get(currentMove);
            currentMove++;
            // Get the color of new location
            String color = getColor(lastX, lastY);
            colorPath.add(color);
            // Call on filtering again
            predict(color, filterDist);
        }
        // Finished with moves, got entire color sequence
        if (currentMove >= xsearchPath.size() && animationDone && !fb) {
            fb = true;
            // Call forward-backward smoothing
            forwardBack(fbDist);
            // Call Viterbi
            Viterbi();
        }
    }
```

The `AnimationPath` class simply deals with the animation of the robot on the colored maze and calls upon the actual filtering algorithm after each time slice (robot moves). The first step in the filtering algorithm is the prediction:

```
public void predict(String color, ArrayList<Double> pDist) {
    // For each of the 16 locations in our vector
    for (int i = 0; i < pDist.size(); i++) {
        double r1 = 0, r2 = 0, r3 = 0, r4 = 0, r5 = 0, r6 = 0;
        double blocked = 0;
        // Get x and y coordinates of the location
        int xcor, ycor;
        int tempcord;
        List<Integer> blah = intcor.get(i);
        xcor = blah.get(0);
        ycor = blah.get(1);
        // Check for walls of all possible adjacent locations
        if (maze.isLegal(xcor, ycor + 1)) {
            // If the adjacent spot is open, get the transitional probability
            tempcord = intrep.get(Arrays.asList(xcor, ycor + 1));
            r1 = pDist.get(tempcord) * .25;
        } else {
            // There is a wall
            blocked++;
        }
        if (maze.isLegal(xcor, ycor - 1)) {
            tempcord = intrep.get(Arrays.asList(xcor, ycor - 1));
            r2 = pDist.get(tempcord) * .25;
        } else {
            blocked++;
        }
        if (maze.isLegal(xcor + 1, ycor)) {
```

```
            tempcord = intrep.get(Arrays.asList(xcor + 1, ycor));
            r3 = pDist.get(tempcord) * .25;
        } else {
            blocked++;
        }
        if (maze.isLegal(xcor - 1, ycor)) {
            tempcord = intrep.get(Arrays.asList(xcor - 1, ycor));
            r4 = pDist.get(tempcord) * .25;
        } else {
            blocked++;
        }
        // Same location, robot moved into a wall
        tempcord = intrep.get(Arrays.asList(xcor, ycor));
        r5 = pDist.get(tempcord) * (blocked * .25);
        // Sum of all possible adjacent locations probabitlies for single cell
        r6 = r1 + r2 + r3 + r4 + r5;
        // Update step
        update(i, r6, color, pDist);
    }
    // Normalize the probability distribution
    normalize(pDist);
}
```

Within the step, we go through each of the $maze.height x maze.width$ locations in the vector and calculate the transitional probabilities. We then need to go through the update step using the provided color evidence. Note that the time and space requirements for updating must be constant if an agent with limited memory is to keep track of the current state distribution over an unbounded sequence of observations.

```
public void update(int indexx, double r6, String color,
        ArrayList<Double> pDist) {
    // Find the actual color of the location
    int xcor, ycor;
    List<Integer> blah = intcor.get(indexx);
    xcor = blah.get(0);
    ycor = blah.get(1);
    // Check if the color matches
    if (getRealColor(xcor, ycor).equals(color)) {
        // 88% chance given the sensored color matches
        pDist.set(indexx, (r6 * .88));
    } else {
        // Doesnt match
        pDist.set(indexx, (r6 * .04));
    }
}
```

Once we finished both the prediction and update steps, we can normalize our probability distribution:

```
public void normalize(ArrayList<Double> pDist) {
    // Add all probabilities
    double sum = 0;
    for (double x : pDist) {
        sum = sum + x;
    }
```

```
        // Divide to normalize
        for (int i = 0; i < pDist.size(); i++) {
            pDist.set(i, (pDist.get(i) / sum));
        }
    }
```

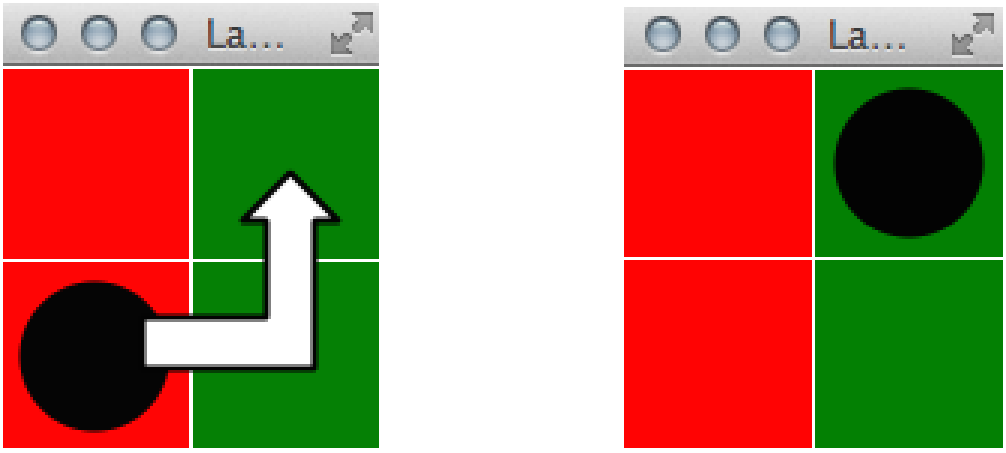
There are two utility functions that we used in our implementation, `getColor` and `getRealColor`. `getColor` is given a location and determines the color of that location and using the random number generator, returns the correct color 88% of the time and a different color (one of the other three colors) the other 12% of the time. `getRealColor` returns the correct color for a given square. The implementation are as follows:

```
    public String getRealColor(int x, int y) {
        // Check to see which color the square is
        String correctColor = null;
        if (maze.isr(x, y))
            correctColor = "r";
        if (maze.isy(x, y))
            correctColor = "y";
        if (maze.isg(x, y))
            correctColor = "g";
        if (maze.isb(x, y))
            correctColor = "b";
        return correctColor;
    }
    public String getColor(int x, int y) {
        String correctColor = null;
        String wrongColor = null;
        // Get the correct color
        correctColor = getRealColor(x, y);
        // Determine what color to return
        double chance = Math.random() * 100;
        // 88% chance to get correct color
        if (chance < 88) {
            return correctColor;
        } else {
            double chance2 = Math.random() * 100;
            // 12% chance to return one of the other colors
            if (chance2 < 33) {
                if (correctColor.equals("r"))
                    wrongColor = "b";
                if (correctColor.equals("g"))
                    wrongColor = "y";
                if (correctColor.equals("y"))
                    wrongColor = "g";
                if (correctColor.equals("b"))
                    wrongColor = "r";
            }
            if (chance2 >= 33 && chance2 <= 67) {
                if (correctColor.equals("r"))
                    wrongColor = "y";
                if (correctColor.equals("g"))
```

```
                wrongColor = "b";
            if (correctColor.equals("y"))
                wrongColor = "r";
            if (correctColor.equals("b"))
                wrongColor = "g";
        }
        if (chance2 > 67) {
            if (correctColor.equals("r"))
                wrongColor = "g";
            if (correctColor.equals("g"))
                wrongColor = "r";
            if (correctColor.equals("y"))
                wrongColor = "b";
            if (correctColor.equals("b"))
                wrongColor = "y";
        }
    }
    return wrongColor;
}
```

2.2 Testing and Results

In order to run test the filtering algorithm, I simply needed to change the `simple.maz` each time I want to change the map. After each movement by the robot, I simply call `predict` and then print out the probability distribution. In my initial test, shown in Figure 1, I get the resulting probability distribution:



(a) Starting Location

(b) Final Position

Figure 1: Initial Test

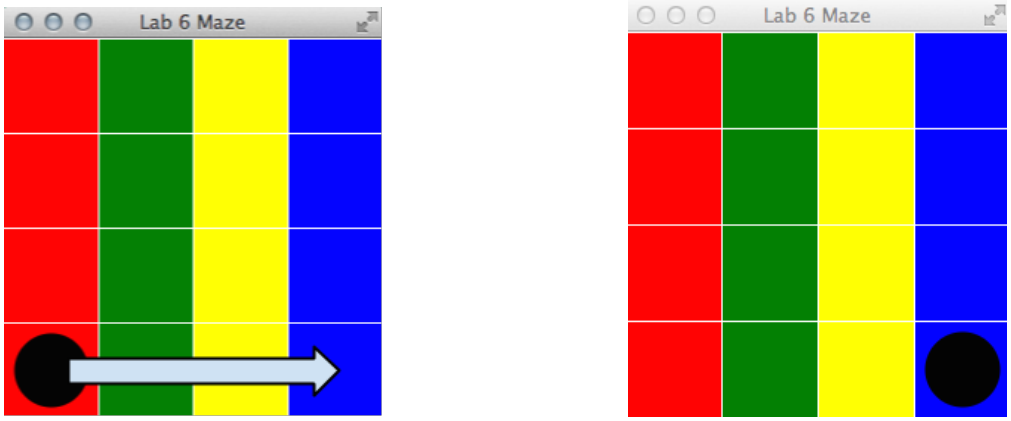
```
1
[0 ,  0]0.48852954826560907
[0 ,  1]0.47387366181764085
[1 ,  0]0.021539711900801856
[1 ,  1]0.016057078015948273

2
[0 ,  0]0.27495997592858307
[0 ,  1]0.1827241938058231
[1 ,  0]0.30343207936269595
[1 ,  1]0.23888375090289796

3
[0 ,  0]0.029371532980112646
[0 ,  1]0.01742581517784261
[1 ,  0]0.5339326553704015
[1 ,  1]0.4192699964716432
```

This is a simple test to see how well our filtering algorithm works. We see that when the robot started off on (0,0), the two coordinates (0,0) and (0,1) have the highest probability, which is correct as those are the cells colored red. Then the robot traveled to (1,0), which is green. The probability distribution reflects this, but doesn't eliminate the possibility that the robot could still be on a red cell because the robot's sensor could be wrong. Once the robot moved onto (1,1), the probability distribution changes again. Now the probability of the robot being on one of the green squares is over 90%, which is quite accurate.

In another test, we increase the size of the maze and the number of colors. The maze and resulting probability distributions are as follows:



(a) Starting Location (b) Final Position

Figure 2: Second Test

1
[0 , 0]0.23549880630857115
[0 , 1]0.228433842119314
[0 , 2]0.22687954999767743
[0 , 3]0.22653760573091739
[1 , 0]0.010383356459968818
[1 , 1]0.0077404175837149005
[1 , 2]0.007698445273735995
[1 , 3]0.007694606107968605
[2 , 0]0.008132201961482795
[2 , 1]0.005510971793374049
[2 , 2]0.005484339768593173
[2 , 3]0.005484035056687689
[3 , 0]0.008109690416497936
[3 , 1]0.005488452220020791
[3 , 2]0.005461973517808211
[3 , 3]0.005461705683667031

2
[0 , 0]0.15584205418473232
[0 , 1]0.10322332773107569
[0 , 2]0.10227160070570428
[0 , 3]0.10218616560508394
[1 , 0]0.16110604201406367
[1 , 1]0.12195617249886141
[1 , 2]0.11298664027303691
[1 , 3]0.11099307565135005
[2 , 0]0.006386969549384329
[2 , 1]0.0036925421021334268
[2 , 2]0.0035700221233531584
[2 , 3]0.0035488024735279918
[3 , 0]0.004829892102351144
[3 , 1]0.00248942444362372
[3 , 2]0.002458922281827223
[3 , 3]0.0024583462598905116

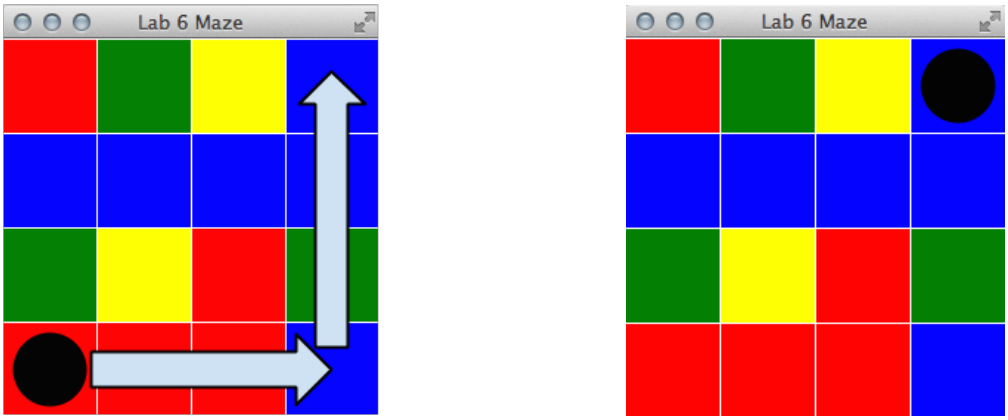
3
[0 , 0]0.17454997125371383
[0 , 1]0.10097335188432476
[0 , 2]0.09720524655653388
[0 , 3]0.096537564026636
[1 , 0]0.08945759096925401
[1 , 1]0.03726172501567716
[1 , 2]0.036060845956536115
[1 , 3]0.03603573012779876
[2 , 0]0.11907694128392486
[2 , 1]0.07479091711072562
[2 , 2]0.06443898546789147
[2 , 3]0.062152194868387356
[3 , 0]0.0048723569119059834
[3 , 1]0.0022961354628154257

```
[3 ,  2]0.0021574362771305203
[3 ,  3]0.002133006826744159

4
[0 ,  0]0.17983664865890492
[0 ,  1]0.08027542846472074
[0 ,  2]0.07740100021084718
[0 ,  3]0.07714133210467972
[1 ,  0]0.08372727242967738
[1 ,  1]0.038589180632667384
[1 ,  2]0.034650173853927664
[1 ,  3]0.033845953155627606
[2 ,  0]0.06708144010049016
[2 ,  1]0.023300883179692998
[2 ,  2]0.022015222325673146
[2 ,  3]0.02198618067605969
[3 ,  0]0.10305405132019747
[3 ,  1]0.06045634415516568
[3 ,  2]0.04960577295437173
[3 ,  3]0.047033115777296526
```

With this maze, both the number of colors and the number of cells are increased. Notice that the group of probabilities for the x coordinate group of the correct color at each time step increases when the robot actually moves onto that color. What is interesting is why the initial coordinate (0,0) remains much higher than the other locations, presumably due to the error from the sensors.

Another test is presented below:



(a) Starting Location (b) Final Position

Figure 3: Third Test

1
[0 , 0]0.21053960352332957
[0 , 1]0.009282882518983166
[0 , 2]0.007270315308939703
[0 , 3]0.15950417201046393
[1 , 0]0.20422341541762967
[1 , 1]0.00692005396853271
[1 , 2]0.004926894681941305
[1 , 3]0.006429301656090634
[2 , 0]0.2028338540343757
[2 , 1]0.15141566152230462
[2 , 2]0.00634841655120904
[2 , 3]0.004912768171239579
[3 , 0]0.009205825024093627
[3 , 1]0.006391205854630564
[3 , 2]0.0049123872132249774
[3 , 3]0.0048832425430112274

2
[0 , 0]0.292927720403924
[0 , 1]0.003421793835705069
[0 , 2]0.003636862644295898
[0 , 3]0.15102369184150538
[1 , 0]0.25553802152401495
[1 , 1]0.005869983412335312
[1 , 2]3.631710134808018E-4
[1 , 3]0.0017518483544061152
[2 , 0]0.22399109756466687
[2 , 1]0.05645010877392767
[2 , 2]7.742845691544728E-4
[2 , 3]2.3080158597471272E-4
[3 , 0]0.0027603254095255455
[3 , 1]8.29276746909745E-4
[3 , 2]2.215678766246892E-4
[3 , 3]2.0944444354875985E-4

3
[0 , 0]0.35398667584063204
[0 , 1]0.0037861049004259944
[0 , 2]0.0029904341312307204
[0 , 3]0.12795304020087364
[1 , 0]0.2812645116743446
[1 , 1]0.003932568669877628
[1 , 2]1.1734261237976172E-4
[1 , 3]0.0013184652664323345
[2 , 0]0.18054267937138282
[2 , 1]0.041256464002099674
[2 , 2]4.223538692690615E-4
[2 , 3]2.5793092262611798E-5
[3 , 0]0.0019263671539390802
[3 , 1]4.5184263865165943E-4

[3 , 2]1.695100128030142E-5
[3 , 3]8.405574917854057E-6

4
[0 , 0]0.16605254881002768
[0 , 1]0.003451292565398431
[0 , 2]0.4829083913003303
[0 , 3]0.04784193434773703
[1 , 0]0.07954109701022283
[1 , 1]0.007748413541452376
[1 , 2]0.11434865668500412
[1 , 3]0.001846691548797446
[2 , 0]0.03820658375593822
[2 , 1]6.057323038556047E-4
[2 , 2]0.025447213388214256
[2 , 3]2.78657718841102E-4
[3 , 0]0.02424121062352769
[3 , 1]3.2686066781900533E-4
[3 , 2]0.005763578562756568
[3 , 3]0.00139113717007729

5
[0 , 0]0.03187725586390654
[0 , 1]0.8417996092890937
[0 , 2]0.05795813052324752
[0 , 3]0.008069404104355348
[1 , 0]0.009956198058829864
[1 , 1]0.01734543726695051
[1 , 2]0.0028490608720803316
[1 , 3]0.0059928010944471375
[2 , 0]0.00494173391659522
[2 , 1]0.002202182063459298
[2 , 2]5.145236987078219E-4
[2 , 3]1.9330452023795866E-4
[3 , 0]0.0037977078398996845
[3 , 1]0.011609661707077164
[3 , 2]6.706855656781738E-4
[3 , 3]2.223036154334687E-4

6
[0 , 0]0.18788947412461296
[0 , 1]0.19009526988096212
[0 , 2]0.3528017477300474
[0 , 3]0.008070070523408978
[1 , 0]0.008496176452578212
[1 , 1]0.0030225771410345
[1 , 2]0.10766220253700051
[1 , 3]0.0024268923974931746
[2 , 0]0.0023305027516290246
[2 , 1]0.0025417679849171052
[2 , 2]0.02814582587898753

```
[2,  3]3.9102211874330077E-4
[3,  0]0.0872244602626255
[3,  1]0.003417948447545672
[3,  2]0.010975914876921958
[3,  3]0.004508146891492215

7
[0,  0]0.027665792188291044
[0,  1]0.026572055808637567
[0,  2]0.502340848261259
[0,  3]0.005917729801517794
[1,  0]9.437373041483509E-4
[1,  1]0.00558337049838368
[1,  2]0.14414055215852267
[1,  3]0.0016363137662958704
[2,  0]0.004445475589974458
[2,  1]0.001620625911128244
[2,  2]0.04411274743098633
[2,  3]6.934696822919839E-4
[3,  0]0.1894595374056529
[3,  1]0.002604113188354528
[3,  2]0.026685822396102746
[3,  3]0.015577808608453024
```

What is interesting about this map is that the probability distributions around the areas with cells that are red, green, and mostly blue are the highest as the path that the robot took (as can been seen in Figure 3) consists of initially red, green, and then blue. This color sequence is most likely why the probabilities for the cells after red and green cells are the highest.

3 Forward-Backward Smoothing

Smoothing is the process of computing the distribution over past states given evidence up to the present, that is, $P(X_k|e_{1:t})$, which is the normalized vector produced by the point wise multiplication of the forward message (essentially our filtering algorithm) and a backward message. This backward message recursively calculated: $P(e_{K+1:t}|X_k)$, which is analogous to the forward message. This calculation is done recursive as can be seen from the expanded equation in Equation 15.9 in the Russell and Norvig text.

In terms of run time, both the forward and backward recursions take a constant amount of time per step. Thus, the time complexity of smoothing with respect to evidence $e_{1:t}$ is $O(t)$. This is the complexity for smoothing at a particular time step k. To smooth the whole sequence, one method is simply to run the whole smoothing process once for each time step to be smoothed, which results in a time complexity of $O(t^2)$. Using dynamic programming would have been able to reduce the complexity to $O(t)$.

3.1 Implementation

After the animation is completed and the entire color sequence is provided, we call on `forwardBack` to run the forward backward smoothing algorithm:

```
public ArrayList<Double> forwardBack(ArrayList<Double> pDist) {
    // Backward message
    ArrayList<Double> b = new ArrayList<Double>();
    // Smoothed
    ArrayList<Double> sv = new ArrayList<Double>();
```

```
        // f[0]
        for (int i = 0; i < maze.width * maze.height; i++) {
            pDist.add((double) 1 / 16);
        }
        // b is filled with 1 initially
        for (int i = 0; i < maze.width * maze.height; i++) {
            b.add((double) 1);
        }
        // Get the forward message and record each time slice in a hashmap
        ArrayList<Double> temp1 = new ArrayList<Double>();
        temp1.addAll(pDist);
        forwards.put(0, temp1);
        for (int j = 0; j < colorPath.size(); j++) {
            predict(colorPath.get(j), pDist);
            ArrayList<Double> temp = new ArrayList<Double>();
            temp.addAll(pDist);
            forwards.put(j + 1, temp);
        }
        // Calculate the backward message
        for (int k = colorPath.size() - 1; k > 0; k--) {
            sv = pointMult(forwards.get(k), b);
            System.out.println(k);
            System.out.println(sv);
            normalize(sv);
            b = backward(b, k);
        }
        return sv;
    }
```

Since the forward message had already been implemented previously, we could simply reuse that code. However, we need to implement the backward message:

```
public ArrayList<Double> backward(ArrayList<Double> b, int time) {
        ArrayList<Double> bb = new ArrayList<Double>();
        // For each location
        for (int i = 0; i < b.size(); i++) {
            double firstarg = 0, thirdarg = 0;
            // Recursively calculate the second argument of 15.9
            firstarg = rbackward(time, i);
            ArrayList<Double> temp = new ArrayList<Double>();
            // We have the forward message at each time saved in hash map
            temp = forwards.get(time - 1);
            thirdarg = temp.get(i);
            // We are changing the backward message for each location
            bb.add(firstarg * thirdarg);
        }
        return bb;
    }

    // To help in recursively calculating the backward message
    public Double rbackward(int time, int index) {
        double firstarg = 0;
```

```
        // Base case
        if (time == colorPath.size() - 1) {
            int xcor, ycor;
            List<Integer> blah = intcor.get(index);
            xcor = blah.get(0);
            ycor = blah.get(1);
            if (getRealColor(xcor, ycor).equals(colorPath.get(time))) {
                firstarg = .88;
            } else {
                firstarg = .04;
            }
            ArrayList<Double> temp = new ArrayList<Double>();
            temp = forwards.get(time - 1);
            firstarg = firstarg * temp.get(index);
            return firstarg;
        } else {
            // Recursive case
            int xcor, ycor;
            List<Integer> blah = intcor.get(index);
            xcor = blah.get(0);
            ycor = blah.get(1);
            if (getRealColor(xcor, ycor).equals(colorPath.get(time))) {
                firstarg = .88;
            } else {
                firstarg = .04;
            }
            ArrayList<Double> temp = new ArrayList<Double>();
            temp = forwards.get(time - 1);
            firstarg = firstarg * rbackward(time + 1, index) * temp.get(index);
            return firstarg;
        }
    }
```

This ends the implementation of the forward-backward smoothing algorithm.

3.2 Results

Using the same test as presented in Figure 2, we get the following smoothed probability distributions:

```
1
[0 ,  0]0.3603286496479794
[0 ,  1]0.21782473215057066
[0 ,  2]0.21144095387369433
[0 ,  3]0.21031053166957292
[1 ,  0]3.192810320539652E-5
[1 ,  1]1.001255600001115E-5
[1 ,  2]9.126077971750293E-6
[1 ,  3]8.951649402265913E-6
[2 ,  0]1.3377963085238752E-5
[2 ,  1]2.40702553673 26E-6
[2 ,  2]2.2935839754897523E-6
[2 ,  3]2.2795712695483014E-6
[3 ,  0]1.0032771844272469E-5
```

```
[3 ,  1]1.60294748125327 48E-6
[3 ,  2]1.56050172587414 92E-6
[3 ,  3]1.55990666808880 05E-6

2
[0 ,  0]0.2390882437855844
[0 ,  1]0.06947647842376999
[0 ,  2]0.0675724070837223
[0 ,  3]0.06740320368695978
[1 ,  0]0.2641433699633145
[1 ,  1]0.1145817860029604
[1 ,  2]0.09111408414648538
[1 ,  3]0.08637576118886534
[2 ,  0]1.64584437109957 76E-5
[2 ,  1]3.18039159525629 03E-6
[2 ,  2]2.87419978100904 03E-6
[2 ,  3]2.82325243366358 6E-6
[3 ,  0]1.56581048626997 73E-4
[3 ,  1]2.14399495540640 9E-5
[3 ,  2]2.06614747874756 05E-5
[3 ,  3]2.06469578484696E-5

3
[0 ,  0]0.17454997125371385
[0 ,  1]0.10097335188432477
[0 ,  2]0.0972052465565339
[0 ,  3]0.09653756402663602
[1 ,  0]0.08945759096925403
[1 ,  1]0.03726172501567717
[1 ,  2]0.03606084595653612
[1 ,  3]0.03603573012779877
[2 ,  0]0.11907694128392488
[2 ,  1]0.07479091711072564
[2 ,  2]0.06443898546789148
[2 ,  3]0.06215219486838736
[3 ,  0]0.004872356911905984
[3 ,  1]0.002296135462815426
[3 ,  2]0.0021574362771305208
[3 ,  3]0.0021330068267441595
```

From these results we see that while the probability distributions at the latter moves are similar to the ones in the regular filtering algorithm, we get a smoothed distribution at $t = 1$ and 2. At these two time steps, we noticed the the probability distributions now indicated more accurately where the robot could have been in addition to lowering the probability of where the robot could not have been by looking at the probability distributions, specifically at $x = 0$ and 1.

4 Viterbi

While the two previously mentioned algorithms determines the most probabilistic location at each time step, we also need a method that will help us find the most likely sequence of movements for the robot. While it may be easy to simply choose the location with the highest probability from the probability distribution at each time step t, there is a problem in that the locations chosen this way may not be adjacent, which would

mean that the path chosen is illegal. The probability of the robot moving through a sequence that results in an illegal path is 0%. Thus, we have the Viterbi algorithm to help us determine the most probable sequence of moves the robot took from start to finish.

In order to implement the Viterbi algorithm, we need to view each sequence as a path through a graph whose nodes are the possible states at each time step. There is a recursive relationship between most likely paths to each state x_{t+1} and most likely paths to each state x_t. This relationship can be represented as an equation connecting the probabilities of the paths: $max_{x1...xt}P(x_1, ..., x_t, X_{t+1}|e_{1:t+1})$. When we expand this equation (as can be seen in 15.11) we notice that it is identical to the filtering equation except the forward message is replaced with the probabilities of the most likely path to each state x_t and the instead of summing over x_t, we maximize over x_t.

In terms of time complexity, it is linear in t, the sequence length, like the filtering algorithm. Unlike filtering, which uses constant space, the space requirement is also linear in t because the Viterbi algorithm needs to keep the pointers that identify the best sequence leading to each state.

4.1 Implementation

The implementation is all contained within one function `viterbi`:

```
public void viterbi() {
    // Initial Reading, t = 1
    ArrayList<Double> tempvitprobs = new ArrayList<Double>();
    for (int h = 0; h < viterbiProbs.size(); h++) {
        int x1cor, y1cor;
        List<Integer> x1 = intcor.get(h);
        x1cor = x1.get(0);
        y1cor = x1.get(1);
        if (getRealColor(x1cor, y1cor).equals(colorPath.get(0))) {
            tempvitprobs.add(.88);
        } else {
            tempvitprobs.add(.04);
        }
    }
    viterbiProbs.clear();
    viterbiProbs.addAll(tempvitprobs);
    // From t = 2
    for (int i = 1; i < colorPath.size() - 1; i++) {
        // To hold the probabilities temporarily
        ArrayList<Double> tempprobs = new ArrayList<Double>();
        // Each location
        for (int j = 0; j < viterbiProbs.size(); j++) {
            // Probability of color given location
            double firstarg = 0;
            int xcor, ycor;
            List<Integer> x1 = intcor.get(j);
            xcor = x1.get(0);
            ycor = x1.get(1);
            if (getRealColor(xcor, ycor).equals(colorPath.get(i + 1))) {
                firstarg = .88;
            } else {
                firstarg = .04;
            }
            // Check all adjacent squares
```

```
double max = 0;
int bestmove = 0;
double r1 = 0, r2 = 0, r3 = 0, r4 = 0, r5 = 0;
double blocked = 0;
int tempcord;
if (maze.isLegal(xcor, ycor + 1)) {
    // Get the integer representation of that square
    tempcord = intrep.get(Arrays.asList(xcor, ycor + 1));
    // Get the previous max
    r1 = .25 * viterbiProbs.get(tempcord);
    if (r1 > max) {
        // If this is best move
        max = r1;
        bestmove = tempcord;
    }
} else {
    blocked++;
}
if (maze.isLegal(xcor, ycor - 1)) {
    tempcord = intrep.get(Arrays.asList(xcor, ycor - 1));
    r2 = .25 * viterbiProbs.get(tempcord);
    if (r2 > max) {
        max = r2;
        bestmove = tempcord;
    }
} else {
    blocked++;
}
if (maze.isLegal(xcor + 1, ycor)) {
    tempcord = intrep.get(Arrays.asList(xcor + 1, ycor));
    r3 = .25 * viterbiProbs.get(tempcord);
    if (r3 > max) {
        max = r3;
        bestmove = tempcord;
    }
} else {
    blocked++;
}
if (maze.isLegal(xcor - 1, ycor)) {
    tempcord = intrep.get(Arrays.asList(xcor - 1, ycor));
    r4 = .25 * viterbiProbs.get(tempcord);
    if (r4 > max) {
        max = r4;
        bestmove = tempcord;
    }
} else {
    blocked++;
}
// Same location, hit wall
tempcord = intrep.get(Arrays.asList(xcor, ycor));
r5 = viterbiProbs.get(tempcord) * (blocked * .25);
```

```
                if (r5 > max) {
                    max = r5;
                    bestmove = tempcord;
                }
                // Updates to prevent mix up of data
                tempprobs.add(firstarg * max);
                ArrayList<Integer> tempBest = new ArrayList<Integer>();
                tempBest.addAll(bestPath.get(bestmove));
                tempBest.add(j);
                tempbestPath.put(j, tempBest);
            }
            bestPath.clear();
            for (int loc = 0; loc < viterbiProbs.size(); loc++) {
                bestPath.put(loc, tempbestPath.get(loc));
            }
            tempbestPath.clear();
            // Normalize and insert into pDist
            normalize(tempprobs);
            viterbiProbs.clear();
            viterbiProbs.addAll(tempprobs);
            tempprobs.clear();
        }
        // Append the final location and print solution
        double max = 0;
        int bestFinal = 0;
        for (int i = 0; i < viterbiProbs.size(); i++) {
            if (viterbiProbs.get(i) > max)
                max = viterbiProbs.get(i);
            bestFinal = i;
        }
        bestPath.get(bestFinal).add(bestFinal);
        System.out.print(bestPath.get(bestFinal));
    }
```

4.2 Results

Using the same test as from Figure 2, we get following result:

[(0,3), (1,3), (2,3), (3,3)]

This is an interesting result because, while the robot traveled across the horizontal axis at $y = 0$, the algorithm did determine a path that followed the correct color sequence, just at a different y value, 3.

5 Acknowledgments

This report was completed with the help from the wonderful TAs and other students through the discussions on the Piazza forum as well as the Russell and Norvig textbook, Artificial Intelligence, A Modern Approach