

Probabilistic Reasoning

Yaozhong Kang

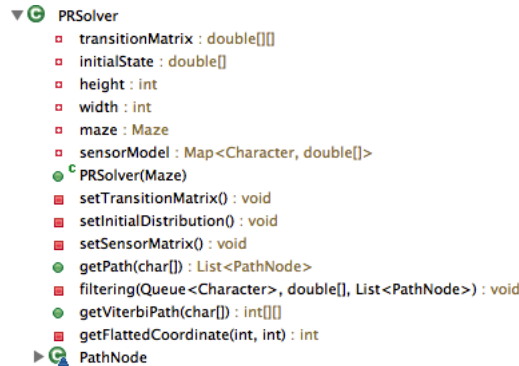
March 9, 2014

1 Introduction

This report mainly discusses the solution to a probabilistic reasoning problem raised by Professor Rob Schapire at Princeton. Section 2 talks about the implementation of the solver. In section 3, there are some experiments done with the solver.

2 Code Design

The major functions are encapsulated in *PRSolver.java*. The constructor of the class accepts a *Maze* object as parameter from which we can get the size of the maze and the color of each grid in the maze. The constructor then calls some private functions to initialize the initial state, the transition matrix and observation matrix. In the following sections, I will talk in detail about the local variables and functions in the class.



```
▼ PRSolver
  transitionMatrix : double[][]
  initialState : double[]
  height : int
  width : int
  maze : Maze
  sensorModel : Map<Character, double[]>
  PRSolver(Maze)
  setTransitionMatrix() : void
  setInitialDistribution() : void
  setSensorMatrix() : void
  getPath(char[]) : List<PathNode>
  filtering(Queue<Character>, double[], List<PathNode>) : void
  getViterbiPath(char[]) : int[][]
  getFlattedCoordinate(int, int) : int
  PathNode
```

Figure 1: The outline of PRServer.java

2.1 Definitions

Here are local variables and an inner class defined in *PRSolver*.

```
// transition matrix
private double[][] transitionMatrix;
// the initial distribution
private double[] initialState;

// size of the board
private int height;
private int width;
```

```

// the board
private Maze maze;

// the observation matrices
private Map<Character, double[]> sensorModel;

/**
 * Distribution at a certain step
 */
class PathNode {
    private double[] [] state;

    public PathNode(double[] arg) {
        this.state = Utils.to2DMatrix(arg, height, width);
    }

    public double[] [] getState() {
        return this.state;
    }
}

```

Suppose the size of the board is $h \times w$ where h is the height and w is the width.

Then the distribution of the system in a given time (including the initial state) can be denoted as a $1 \times h \times w$ vector. The possibility that the robot is in grid(y, x) is stored at index $y \times w + x$ in the vector.

The transition matrix is a $h \times w \times h \times w$ matrix, where $transitionMatrix(i, j)$ is the possibility the robot moves from i to j .

The sensor model is given by a hashmap, where the key is a character standing for a color and the value is a vector which is the possibilities the robot senses the color at the grids.

The inner class *PathNode* is used to store the distribution of the system after receiving each input from the sensor.

2.2 Initial Distribution

Here is the code used to initialize the initial distribution:

```

/**
 * set the initial distribution as uniform distribution
 */
private void setInitialDistribution() {
    int numberOfTiles = 0;
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            if (maze.isLegal(x, y)) {
                numberOfTiles++;
            }
        }
    }
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            if (maze.isLegal(x, y)) {
                this.initialState[this.getFlattedCoordinate(x, y)] = (double) 1.0 / numberOfTiles;
            }
        }
    }
}

```

```
}  
}
```

The initial distribution is set to uniform distribution. For example, if there are n legal grids on the board, then the distribution is $\frac{1}{n}$ for legal grids and is 0 for the wall grids.

2.3 Transition Matrix

Following is the code for initializing the transition matrix:

```
/**  
 * initialize transition matrix  
 */  
private void setTransitionMatrix() {  
    int size = this.height * this.width;  
    this.transitionMatrix = new double[size][size];  
    // the possibility of moving towards whatever direction  
    double unit = 0.25;  
    for (int from = 0; from < height; from++) {  
        for (int to = 0; to < width; to++) {  
            if (maze.isLegal(to, from)) {  
                double total = 0.0;  
                if (maze.isLegal(to - 1, from)) {  
                    total += unit;  
                    this.transitionMatrix[this.getFlattedCoordinate(to, from)][this  
                        .getFlattedCoordinate(to - 1, from)] = unit;  
                }  
  
                if (maze.isLegal(to + 1, from)) {  
                    total += unit;  
                    this.transitionMatrix[this.getFlattedCoordinate(to, from)][this  
                        .getFlattedCoordinate(to + 1, from)] = unit;  
                }  
  
                if (maze.isLegal(to, from - 1)) {  
                    total += unit;  
                    this.transitionMatrix[this.getFlattedCoordinate(to, from)][this  
                        .getFlattedCoordinate(to, from - 1)] = unit;  
                }  
  
                if (maze.isLegal(to, from + 1)) {  
                    total += unit;  
                    this.transitionMatrix[this.getFlattedCoordinate(to, from)][this  
                        .getFlattedCoordinate(to, from + 1)] = unit;  
                }  
  
                this.transitionMatrix[this.getFlattedCoordinate(to, from)][this  
                    .getFlattedCoordinate(to, from)] = 1.0 - total;  
            }  
        }  
    }  
}
```

For each grid g_{from} on the board, the program tries to move towards all of the four directions. If the move is legal (the destination grid g_{to} is not a wall or going out of bound), then set $transitionMatrix[g_{from}][g_{to}]$ to

0.25. Finally, the possibility the robot stays where it was can be calculated by $1 - 0.25 * \text{number_of_legal_moves}$ where $0.25 * \text{number_of_legal_moves}$ is stored in the local variable called 'total'.

2.4 Observation Matrix

Following is the code for initializing the observation matrix:

```
/**
 * initialize the observation matrix
 */
private void setSensorMatrix() {
    int size = this.height * this.width;
    char[] colors = new char[] { 'b', 'r', 'g', 'y' };
    for (char c : colors) {
        double[] sensorDistribution = new double[size];
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                double value = maze.isLegal(x, y) ? (maze.getChar(x, y) == c ? 0.88
                    : 0.04)
                    : 0.0;
                sensorDistribution[this.getFlattedCoordinate(x, y)] = value;
            }
        }
        this.sensorModel.put(c, sensorDistribution);
    }
}
```

There are totally 4 colors in the system, with 'b' standing for Blue, 'r' for Red, 'g' for Green and 'y' for Yellow. For each color, compute the corresponding vector and put them into the map. For each grid on the board, if the current color matches its color, the possibility is set to 0.88, otherwise it is set to 0.04.

2.5 Filtering

Here is the code for the filtering part:

```
/**
 * get the distribution of each step
 *
 * @param sensorInput
 * @return
 */
public List<PathNode> getPath(char[] sensorInput) {
    List<PathNode> result = new LinkedList<PathNode>();
    // put the input sequence to a queue
    Queue<Character> input = new LinkedList<Character>();
    for (char c : sensorInput) {
        input.add(c);
    }
    this.filtering(input, this.initialState.clone(), result);
    return result;
}

/**
 * compute the distribution of each step according to the current
 * observation and previous status
```

```

*
* @param input
* @param currentState
* @param path
*/
private void filtering(Queue<Character> input, double[] currentState,
    List<PathNode> path) {
    path.add(new PathNode(currentState));
    if (input.isEmpty()) {
        return;
    }

    char currentInput = input.poll().charValue();

    // get the observation matrix according to the current observation
    double[] sensorMatrix = this.sensorModel.get(currentInput);
    // apply to the transition matrix
    Utils.multiply(currentState, this.transitionMatrix);
    // apply the observation
    Utils.multiply(currentState, sensorMatrix);
    // normalize
    Utils.normalize(currentState);
    // next recursion
    this.filtering(input, currentState, path);
}

```

Here is how it works. *getPath* function puts the sensor's input into a queue and pass the queue into the *filtering* function. Once the queue is not empty, the *filtering* function recursively calls itself and does the following operations: 1. multiply the current distribution matrix with the transition matrix; 2. apply the observation matrix according to the current sensor input to the result got from step 1. Here the *multiply* function used in step 2 is different from the matrix multiplication operation in step 1. For example, for vector1=(a1, a2, a3 ... an) and vector2=(b1, b2, b3 ... bn), the result vector would be (a1*b1, a2*b2, ... an*bn) (See 'Utility Functions' section for codes); 3. normalize the distribution matrix. It also keeps record of the distribution matrix at each input and put them into the result list. This list is used to animate the changes of the distribution over time.

2.6 Viterbi Algorithm (bonus)

Here is the code used to get the most-likely path of the robot by Viterbi algorithm:

```

/**
 * get the Viterbi path according to observation sequence
 *
 * @return int[][] result where:
 *     result[0][k] = coordinate x in step k
 *     result[1][k] = coordinate y in step k
 */
public int[][] getViterbiPath(char[] sensorInput) {
    int num_of_states = this.initialState.length;
    int num_of_steps = sensorInput.length;

    // store the distributions
    double[][] distribution = new double[num_of_states][num_of_steps];
    // store the path
    int[][] vpath = new int[num_of_states][num_of_steps];
}

```

```

// compute the distribution in the first observation
double[] observation = this.sensorModel.get(sensorInput[0]);
for (int i = 0; i < num_of_states; i++) {
    if (initialState[i] == 0.0 || observation[i] == 0.0) {
        distribution[i][0] = Double.NEGATIVE_INFINITY;
        continue;
    }
    distribution[i][0] = Math.log(initialState[i])
        + Math.log(observation[i]);
}

// iteratively compute the distribution for the later observations
for (int o = 1; o < sensorInput.length; o++) {
    // get the observation vector for the current observation
    observation = this.sensorModel.get(sensorInput[o]);
    for (int to = 0; to < num_of_states; to++) {
        double maxPossibility = Double.NEGATIVE_INFINITY;
        int maxIndex = -1;
        for (int from = 0; from < num_of_states; from++) {
            double transition = this.transitionMatrix[from][to];
            double p = transition == 0.0
                || distribution[from][o - 1] == Double.NEGATIVE_INFINITY ? Double.NEGATIVE_INFINITY
                : distribution[from][o - 1] + Math.log(transition);
            if (p > maxPossibility) {
                maxPossibility = p;
                maxIndex = from;
            }
        }
        distribution[to][o] = maxPossibility == Double.NEGATIVE_INFINITY ? Double.NEGATIVE_INFINITY
            : maxPossibility + Math.log(observation[to]);
        vpath[to][o] = maxIndex;
    }
}

double max = Double.NEGATIVE_INFINITY;
int row = 0;
for (int i = 0; i < num_of_states; i++) {
    double p = distribution[i][num_of_steps - 1];
    if (p > max) {
        max = p;
        row = i;
    }
}
int[][] result = new int[2][num_of_steps + 1];
result[0][num_of_steps] = row % this.width;
result[1][num_of_steps] = row / this.width;

// parse the result to a 2D array
for (int i = num_of_steps - 1; i >= 0; i--) {
    result[0][i] = vpath[row][i] % this.width;
    result[1][i] = vpath[row][i] / this.width;
    row = vpath[row][i];
}

return result;

```

}

There are 2 important local variables: *distribution* and *vpath*. Both are of size $n*m$ where n is the number of states and m is the length of the input sequence. In *vpath*, *vpath*[i][j] stores index k which means after knowing the j^{th} input of the sensor, the most-likely preceding position of position i is position k along the path. In *distribution*, *distribution*[i][j] is the possibility the robot is going through a certain most-likely path and stay at position i according to the first j inputs.

At first, we need to compute the initial distribution according to the first sensor input. Then for each later input, and for each grid on the board, the program computes the possibility of all other grids moving to the current grid and selects the grid with the largest possibility as precursor. Then the program records the index of the precursor in *vpath* and records the possibility value in *distribution*. Finally, we check the last column of *distribution* and find the row number r with the largest possibility. Here r means after sensing the input sequence the robot is most-likely to go through a path and at the end stay at grid r . With r , we can do something like a backtracking to restore the complete path from *vpath* by iteratively getting the most-likely precursor.

If the path is long, then we need to deal with smaller and smaller possibility values. To prevent it from overflowing, I used log on the double value manipulations. Instead of doing $a*b$, I used $\log(a)+\log(b)$. Since $a*b > c*d \Leftrightarrow \log(a)+\log(b) > \log(c)+\log(d)$ where $a,b,c,d > 0$, this change does not affect correctness of the program.

The result of the Viterbi algorithm will be shown together with other parts in the 'Experiments' section.

2.7 *Utility Functions

Below is just some utility functions I wrote to support the main functionalities, such as multiplication of the matrices and vectors, format function of a double matrix.

```
public class Utils {
    /**
     * apply vector = vector * matrix
     *
     * @param vector
     *           : 1*n vector
     * @param matrix
     *           : n*n matrix
     */
    public static void multiply(double[] vector, double[][] matrix) {
        int length = vector.length;
        double[] result = new double[length];
        for (int k = 0; k < length; k++) {
            double current = 0.0;
            for (int j = 0; j < length; j++) {
                current += vector[j] * matrix[j][k];
            }
            result[k] = current;
        }
        for (int k = 0; k < length; k++) {
            vector[k] = result[k];
        }
    }

    /**
     * @param vector1
     *           = (a1, a2, a3 ... an)
     * @param vector2
```

```

*           = (b1, b2, b3 ... bn)
*
*           return vector = (a1*b1, a2*b2, ... an*bn)
*/

public static void multiply(double[] vector1, double[] vector2) {
    int length = vector1.length;
    for (int k = 0; k < length; k++) {
        vector1[k] = vector1[k] * vector2[k];
    }
}

/**
 * normalize a vector
 *
 * @param vector
 */
public static void normalize(double[] vector) {
    double total = 0.0;
    int h1 = vector.length;
    for (int i = 0; i < h1; i++) {
        total += vector[i];
    }
    for (int i = 0; i < h1; i++) {
        vector[i] = vector[i] / total;
    }
}

/**
 * transform a 1D vector to a 2D matrix
 *
 * @param vector
 * @param height
 * @param width
 * @return
 */
public static double[][] to2DMatrix(double[] vector, int height, int width) {
    double[][] result = new double[height][width];
    int index = 0;
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            result[i][j] = vector[index++];
        }
    }
    return result;
}

/**
 * transform a 2D double matrix to a string
 *
 * @param matrix
 * @return
 */
public static String matrix2String(double[][] matrix) {
    StringBuffer result = new StringBuffer();
    int height = matrix.length;

```



```

    int width = matrix[0].length;
    NumberFormat formatter = new DecimalFormat("#0.000 ");
    for (int i = height - 1; i >= 0; i--) {
        for (int j = 0; j < width; j++) {
            result.append(formatter.format(matrix[i][j]));
        }
        result.append("\n");
    }
    return result.toString();
}

/**
 * input string shown on the board
 *
 * @param input
 * @return
 */
public static String buildInputString(char[] input) {
    StringBuffer result = new StringBuffer();
    result.append("Input:\n");
    for (char c : input) {
        result.append(c);
        result.append(" ");
    }
    return result.toString();
}

/**
 * used to animate the path
 *
 * @param c
 * @param possibility
 * @return
 */
public static Color getColor(char c, double possibility) {
    possibility = possibility < 0.5 ? possibility * 2 : 1;
    switch (c) {
        case 'r':
            return Color.color(1.0, 0, 0, possibility);
        case 'g':
            return Color.color(0, 1.0, 0, possibility);
        case 'b':
            return Color.color(0, 0, 1.0, possibility);
        case 'y':
            return Color.color(1.0, 1.0, 0, possibility);
        default:
            return Color.LIGHTGREY;
    }
}
}

```

3 Experiments

Before showing the results, I'd like to first explain how the results are presented. Figure 2 is an example screenshot. As we can see, there are two grid views on the top, and two text boxes on the bottom. The

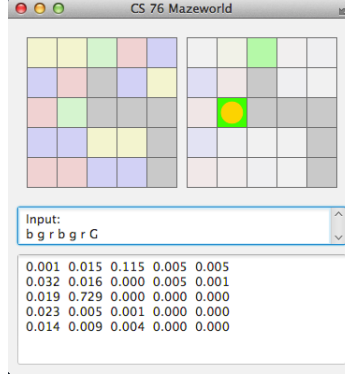


Figure 2: Example.java

upper-left grid view is the initial distribution and it will not change over the time. The upper-right grid view shows the distribution change over time. The possibility values are presented by the value of alpha channel of the color. As a result, the brighter the color is for a grid, the higher the possibility is that the robot is currently at the grid. The golden solid circle will move along the path computed by Viterbi algorithm. The text box in the middle is the input sequence from the sensor. The input which is most-lately handled is shown in upper case while others are in lower case. The text box in the bottom shows the distribution change overtime.

The first experiment is an interesting case, where there is a narrow corridor. The result of this experiment is shown in Figure 3.

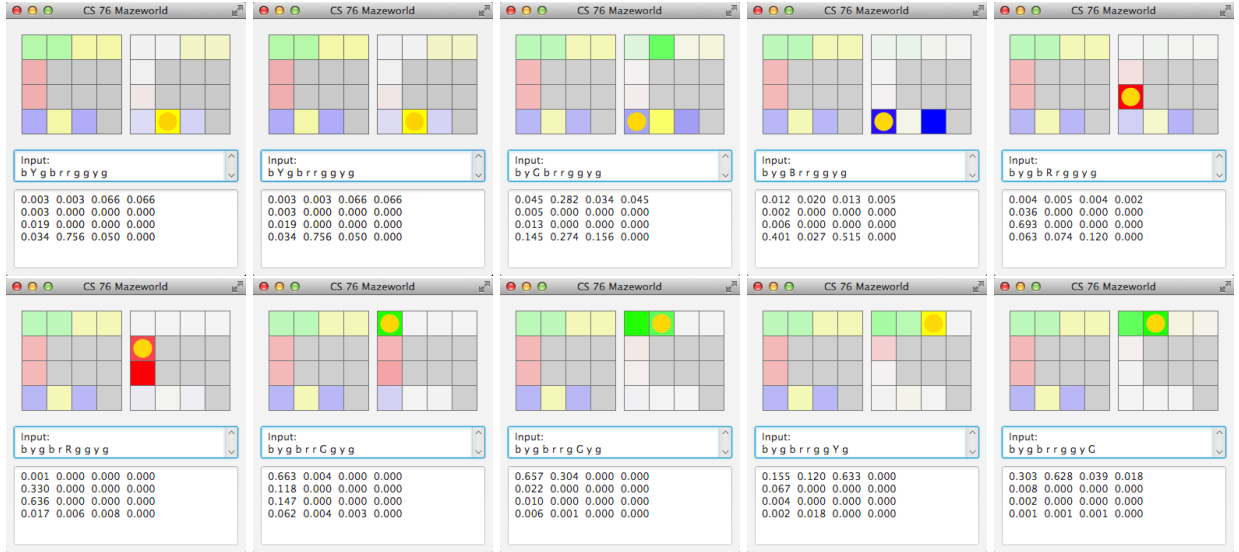


Figure 3: Experiment 1

According to the Viterbi path, we notice that there are some wrong sensor inputs. For example in step 3, the robot is most-likely to be at the left bottom corner. The grid is blue but the sensor's input is green.

Another interesting thing we can see is that, Viterbi algorithm is smarter. It is not a surprise since Viterbi algorithm restores the path after knowing all the sensor inputs while the filtering algorithm only takes into account the sensor's input before the current location. Consider step 6, the filtering algorithm gives the red grid at the bottom a possibility of 0.636 while the above one is 0.330. But the Viterbi algorithm knows the sensor will later give green and then yellow, so it decides that the robot is more likely to be at the above grid. In experiment 2 showing in Figure 4 below, we can see that more clearly.

In experiment 2, I used a map where there are a lot of similar paths. The sensor's input sequence matches each row on the map, except the last one. In this case, before reading the last input, the filtering algorithm will give the columns equal possibility values while Viterbi algorithm will find out which row exactly the path is located.

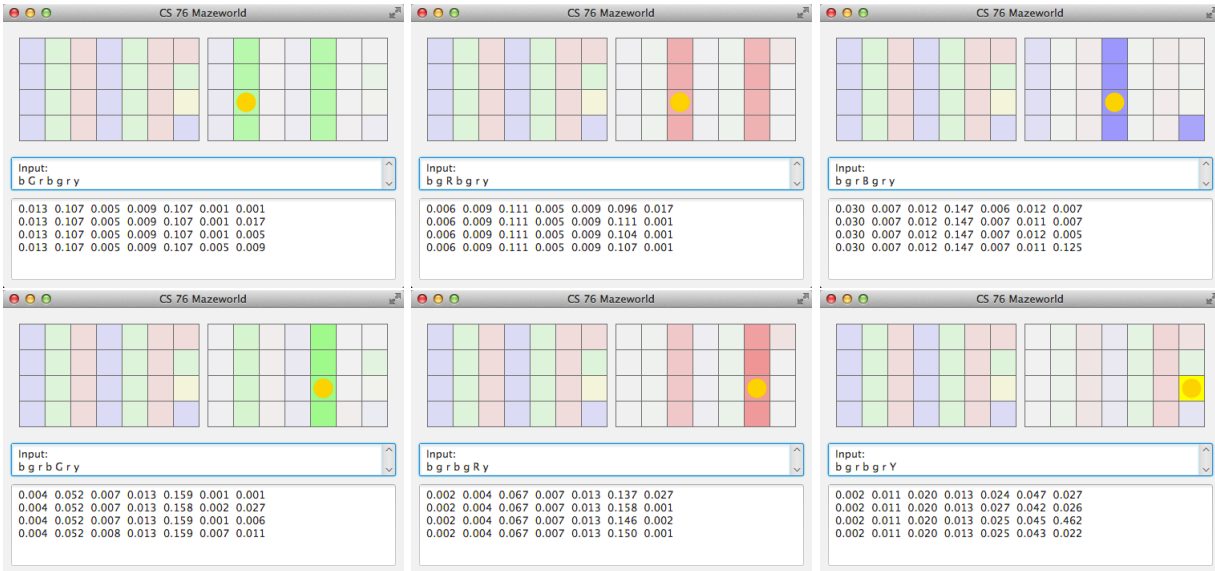


Figure 4: Experiment 2

4 References

- [1] http://en.wikipedia.org/wiki/Viterbi_algorithm
- [2] Answer in 'https://piazza.com/class/hovdpszv7ce5oi?cid=407'
- [3] 'Maze.java' and 'MazeView.java' used in the project are from code provided by Professor Devin Balkcom.