Robot Localisation, Artificial Intelligence

Nils Fagerberg, Daniel Odenbrand March 2, 2015

Background

The purpose of this report is to explain our solution to a project in the course Applied Artificial Intelligence (EDA132) given at Lund University in the spring semester of 2015.

The programming task consisted of implementing a Hidden Markov Model for localising a robot using the robot's own (sometimes inaccurate) sensor. This report will briefly describe and discuss implementation choices and results as well provide instructions on how to run the program yourself.

Robot simulator

We have chosen not to have a graphic representation of the grid and the robot for simplicity. The robot moves on a grid of size 8x8 with a random starting location. Between each time step, the robot moves one step in the north, east, west or south direction and attempts to send a signal of where it is, according to the following rules:

- The robot continues it is heading with a probability of 0.7 and takes another random direction with a probability of 0.3.
- The robot cannot bump into a wall. If it would, instead it turns in another random direction (not resulting in a direction facing another wall).
- The sensor returns the correct location with the probability 0.1, a random location on the inner ring (1 step from the true location) with probability 0.4, a random location on the outer ring (2 steps from the true location) with probability 0.4 and "nothing" with probability 0.1.
- If the robot would return a location outside of the grid, this too is interpreted as "nothing".

Hidden Markov Model

For the Hidden Markov Model we need to keep track of 3 different matrices: the transition model (the probabilities that the robot would transition from one state to another), the evidence matrix (for every move sensed by the sensor, this matrix contains the probabilities that the robot would be in each state) and the belief state (the probabilities for the robot's current position). The transition model will remain constant throughout the simulation, whilst the evidence matrix and the belief state are updated with each time step.

To create the transition model we take a matrix of size 64x64 (technically, transitions between every single state are represented, but most have probabilities 0) and calculate the probabilities given the rules in the previous section. You may see the creation of the matrix in the following code:

Listing 1: Sample java code – Transition Model

```java

    private void setTransitions(int x, int y) {
        //Probability to move to a certain state from the edge is 71/180
        //in the directions along the edge and
        //32/180 in the direction away from the edge
        //From corner 1/2 in each adjacent space
        //Anywhere else is 1/4 for each adjacent space

        //x and y going from 0 to GRID_HEIGHT*GRID_WIDTH
        if (x == 0) {
            if (y == 0) { //Top left corner
                transitionModel[0][1] = 0.5;
                transitionModel[0][8] = 0.5;
            } else if (y == Bot.UPPER_Y_BOUND) { //Bottom left corner
                transitionModel[56][48] = 0.5;
                transitionModel[56][57] = 0.5;
            } else { //Left edge
                int state = 8 * y;
                transitionModel[state][state - 8] = 71.0/180;
                transitionModel[state][state + 8] = 71.0/180;
                transitionModel[state][state + 1] = 32.0/180;
            }
        } else if (x == Bot.UPPER_X_BOUND) {
            if (y == 0) { //Top right corner
                transitionModel[7][6] = 0.5;
                transitionModel[7][15] = 0.5;
            } else if (y == Bot.UPPER_Y_BOUND) { //Bottom right corner
                transitionModel[63][62] = 0.5;
                transitionModel[63][55] = 0.5;
            } else { //Right edge
                int state = 8 * y + 7;
                transitionModel[state][state - 8] = 71.0/180;
                transitionModel[state][state + 8] = 71.0/180;
                transitionModel[state][state - 1] = 32.0/180;
            }
        } else if (y == 0) { //Top edge
            transitionModel[x][x - 1] = 71.0/180;
            transitionModel[x][x + 1] = 71.0/180;
            transitionModel[x][x + 8] = 32.0/180;
        } else if (y == Bot.UPPER_Y_BOUND) { //Bottom ledge
```

```java
            int state = 56 + x;
            transitionModel[state][state - 1] = 71.0/180;
            transitionModel[state][state + 1] = 71.0/180;
            transitionModel[state][state - 8] = 32.0/180;
        } else { //Middle
            int state = y * 8 + x;
            transitionModel[state][state - 8] = 0.25;
            transitionModel[state][state - 1] = 0.25;
            transitionModel[state][state + 1] = 0.25;
            transitionModel[state][state + 8] = 0.25;
        }
    }
```

As you can see, every row in the transitionModel matrix is the state that we currently are in, whilst every column represents the state we could possibly transcend.

We chose to represent the evidence matrix as an 8x8 matrix, while mathematically it ought to be a 64x64 diagonal matrix. While programming, however, it seems unnecessary to store a large number of zeroes for no apparent reason since we need to implement matrix multiplication ourselves anyway. The evidence matrix is simply updated with the probabilities that we are in each state given the sensor's report (represented as a coordinate). Here is the code used:

Listing 2: Sample java code – Evidence Matrix

```java

    private void updateEvidence(Coordinate c) {
        for (int y = 0; y < GRID_WIDTH; ++y) {
            for (int x = 0; x < GRID_HEIGHT; ++x) {
                //probability also handles c == Coordinate.NOTHING
                evidence[y][x] = c.probability(new Coordinate(x, y));
            }
        }
    }


        public double probability(Coordinate c) {
        if (equals(NOTHING)) {
            int x = c.x;
            int y = c.y;
            if (((x == 0 || x == Bot.UPPER_X_BOUND) && (y == 0 || y == Bot.UPPER_Y_BOUND))) { //Corner
                return 0.625;
            } else if (((x == 1 || x == Bot.UPPER_X_BOUND - 1) && (y == 0 || y == Bot.UPPER_Y_BOUND)) ||
                    ((x == 0 || x == Bot.UPPER_X_BOUND) && (y == 1 || y == Bot.UPPER_Y_BOUND - 1))) { //Corner +- 1
                return 0.5;
            } else if (x == 0 || x == Bot.UPPER_X_BOUND || y == 0 || y == ↩
                Bot.UPPER_Y_BOUND) { //Edge, not corner
                return 0.425;
            } else if ((x == 1 || x == Bot.UPPER_X_BOUND - 1) && (y == 1 ↩
                || y == Bot.UPPER_Y_BOUND - 1)) { //Corner 1 in
                return 0.325;
            } else if (x == 1 || y == 1 || x == Bot.UPPER_X_BOUND - 1 || y↩
                 == Bot.UPPER_Y_BOUND - 1) { //Corner 1 in
                return 0.225;
            } else {
                return 0.1;
            }
        }
        if (equals(c)) {
            return 0.1;
        } else if (Math.abs(x - c.x) <= 1 && Math.abs(x - c.x) <= 1) {
            return 0.05;
        } else if (Math.abs(x - c.x) <= 2 && Math.abs(x - c.x) <= 2) {
            return 0.025;
        }
        return 0;
    }
```

If the sensor gave "nothing", we have the giant special case in the function probability. This is due to the previously mentioned fact that the sensor reporting a coordinate outside of the grid is interpreted as "nothing". Thus, some evidence is actually gathered from a "nothing" - the robot is more likely to be close to the edge!

The belief state is also represented as an 8x8 matrix, whilst mathematically it would be convenient to represent it as a 64x1 column vector since we have 64 different states and the algorithm consists of matrix multiplications. Updating the belief state is done according to the forward-algorithm (Russell and Norvig, 2014, *Artificial Intelligence A Modern Approach*):

$$f_{t+1} = \alpha * O_t * T^T * f_t,$$

where alpha is a normalization constant.

If one is interested in the actual code used for the matrix multiplication of our implementation, here it is:

Listing 3: Sample java code – Forward Equation

```

    private void updateBeliefState() {
        //f_t+1 = O_t * T^T * f_t
        double[][] temp  = new double[8][8];
        double OT = 0.0;

        double alpha = 0.0;
        for (int y = 0; y < GRID_HEIGHT*GRID_WIDTH; ++y) {
            double sum = 0.0;
            for (int x = 0; x < GRID_WIDTH*GRID_HEIGHT; ++x) {
                OT = transitionModel[x][y] * evidence[y / GRID_HEIGHT][y % GRID_WIDTH];
                sum += OT * beliefState[x / GRID_HEIGHT][x % GRID_WIDTH];
            }
            temp[y / GRID_HEIGHT][y % GRID_WIDTH] = sum;
            alpha += sum;
        }
        alpha = 1.0/alpha;

        //Normalize
        for (int y = 0; y < GRID_HEIGHT; ++y) {
            for (int x = 0; x < GRID_WIDTH; ++x) {
                beliefState[y][x] = alpha*temp[y][x];
            }
        }
    }
```

We run the algorithm until the model reaches the point where a certain state has more than 99 percent probability to be the correct one. Then we print the true location as well as the location that our model outputs.

Results, discussion and running the code

By running the test a couple of thousand times, the method has about a 60 percent chance to report the exact location for the robot. When it reports a erroneous location, it is usually an adjacent square. Another fairly common misread is that the robot believes that it is somewhere along the edge when in reality, it is, but along another edge. This is most likely due to our representation of reads outside the grid. As previously mentioned, this causes the sensor to report "nothing", thus increasing the probability of the robot being close to an edge - any edge! However, it is difficult to get a higher precision since the grid contains no obstacles apart from the walls surrounding it. When we have achieved our specified accuracy limit, the robot usually believes itself close to an edge. This is also to be expected since once we have reached an edge, continuation along the edge is more likely. Also, sensor reports from the middle doesn't provide as much information as reads of "nothing" combined with reads along the edge.

To run the code go to "/h/d9/q/tpi11dod/Documents/Artificiell\ Intelligens/Handin2" in your terminal. Whilst being in this directory, simply use the command "java -jar Sensorbot.jar". This will execute the program and let you set up your Sensorbot. In this folder you can also view the source code and open it in the editor of your choice by typing for example "emacs Sensor.java". This command would then open Sensor in emacs.