“Robot” localisation with HMM based forward-filtering

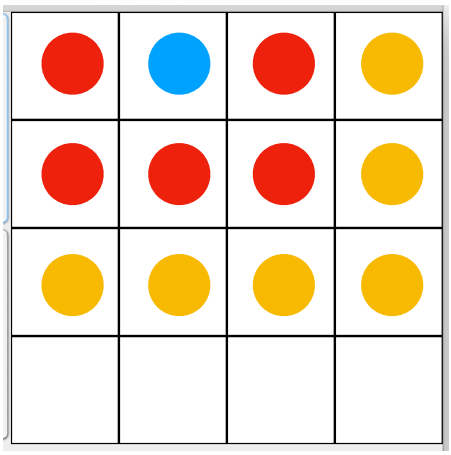
This task is essentially corresponding to task 15.9 in the course book, with a more detailed specification of the sensor model. The task relies on the explanations for matrix based forward filtering operations according to section 15.3.1 of the book.
Hence:
If you do not have access to this particular edition (Artificial Intelligence: A Modern Approach, 3/e, by Stuart Russell and Peter Norvig, ISBN-10: 0132071487) and thus have difficulties in figuring out what to do, please contact me (Elin) at Elin_Anna.Topp@cs.lth.se before attempting to solve the task!

In short

You are assumed to work with robot localisation based on forward filtering with a Hidden Markov Model, read a related article and write a report.

In detail (implementation)

You are supposed to implement an HMM to do forward filtering for localisation in an environment without any landmarks. Consider the (in the book) previously mentioned vacuum cleaner robot *in an empty room*, represented by an *n x m* rectangular grid. The robot’s location is hidden; the only evidence available to you (the observer) is a noisy sensor that gives a direct, but vague, approximation to the robot’s location. The sensor gives approximations $S = (x',y')$ for the (true) location $L = (x, y)$, the directly surrounding fields *Ls* or the “second surrounding ring” *Ls2* according to the specifications below. Here, *n_Ls* is the number of directly surrounding fields for *L* (this can be 3, 5, or 8, depending on whether *L* is in a corner, along a “wall” or at least 2 fields away from any “wall”) and *n_Ls2* is the number of secondary surrounding fields for *L* (this can be 5, 6, 7, 9, 11, or 16, depending on where *L* is located relative to “walls” and corners).



In the example in the figure, the blue spot marks *L*, the red spots are the in this situation possible 5 positions in *Ls,* and the yellow dots mark the 6 possible positions in *Ls2.*

The sensor reports

- the true location *L* (blue) with probability 0.1
- any of the $n\_Ls \in \{3, 5, 8\}$ existing surrounding fields *Ls* (red, here $n\_Ls = 5$) with probability 0.05 each
- any of the $n\_Ls2 \in \{5, 6, 7, 9, 11, 16\}$ existing “secondary” surrounding fields *Ls2* (yellow, here $n\_Ls2 = 6$) with probability 0.025 each
- "nothing" with probability 1.0 - 0.1 - *n_Ls**0.05 - *n_Ls2**0.025.

This means that the sensor is more likely to produce “nothing” when the robot’s true location is less than two steps from a wall or in a corner (there are also other possibilities of setting up the sensor model, but you will stick to this model for the implementation).

The robot moves according to the following strategy:

Pick random start heading h_0. For any new step pick new heading h_t+1 based on the current heading h_t according to:

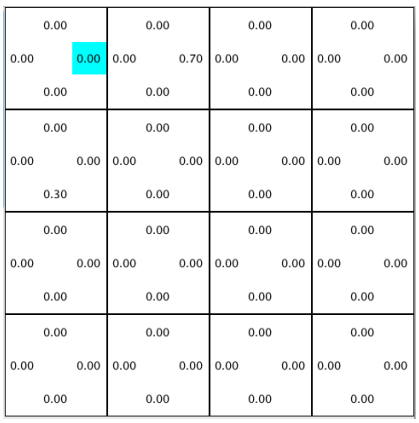
P(h_t+1 = h_t | not encountering a wall) = 0.7
P(h_t+1 != h_t | not encountering a wall) = 0.3
P(h_t+1 = h_t | encountering a wall) = 0.0
P(h_t+1 != h_t | encountering a wall) = 1.0

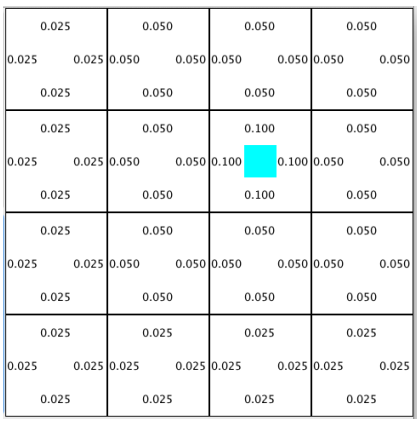
It then moves in the direction h_t+1 by one step in the grid. This means essentially that it a) will always move one step and b) it can only move straight.

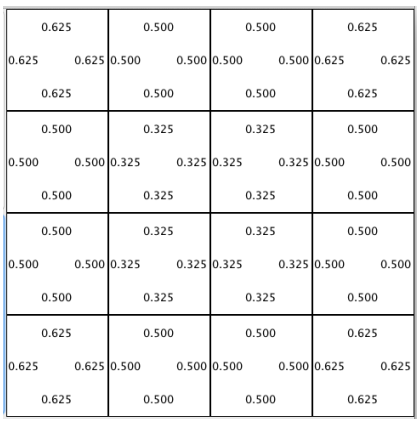
In case a new heading is to be found, the new one is randomly chosen from the possible ones (facing a wall somewhere along the wall leaves three, facing the wall in a corner leaves two options of where to turn, see one example below in hint 1).

Implement this as an HMM (according to the matrix-vector notation suggested in section 15.3.1 of the course book) and apply simple **forward filtering** to track the robot. This requires obviously a two-part implementation, as you need to simulate the robot and its movement (from which you also can simulate the sensor readings) to have some ground truth to evaluate your tracking against, and the HMM-based tracking algorithm as such. Your algorithm should basically loop over the following three steps:
1) move (simulated) robot to new pose according to movement model
2) obtain (simulated) sensor reading based on its actual position given the sensor model
3) update the position estimate (vector f) using the forward-algorithm based on the sensor reading from step 2, using the known sensor and transition models.
Implement also some evaluation (see below).

Hint 1: The handout contains a state model with the following encoding of a state: Each possible state represents the position of the robot in the grid plus a heading. This means you will have rows * columns * 4 possible states, and rows*columns+1 possible sensor readings.
Your transition matrix will thus have the dimensions (rows*columns*4) x (rows*columns*4) and there are (rows*columns+1) observation matrices, each being a diagonal matrix of dimensions (rows*columns*4) x (rows*columns*4) to get the matrix multiplications to work out (the model stores them as vectors to save space). Headings run clockwise from NORTH (0) to WEST (3).
The handout code contains also implementations of the transition and sensor models. When starting to work with the handout, make sure you get the following results when visualising the transition model (left) and the sensor model (centre and right) for a 4x4-grid, details below.







In the three figures above you see a grid visualisation (each cell (x, y) has four headings, representing a bundle of four states with DIFFERENT probabilities to be reached from a specific state, but with the SAME probability to have caused a certain sensor reading) of one row of the transition matrix and the diagonal of one observation matrix. The images show the following:

Left: The transition visualisation shows the probabilities to go from the current state (cyan) to any other state (pose).

Centre: Here, you see the probabilities for each respective state / pose (x, y, h) to have generated the (cyan) reading, expressed as position (x,y).

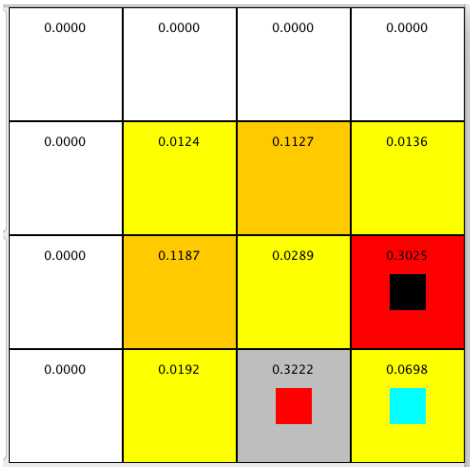
Right: These are the probabilities for each respective state (x, y, h) to have generated “nothing”.

Hint 2: Even though a sensor reading of “nothing” normally means to do the forward step without update, i.e. it boils down to mere prediction in theory, you should use the information given in the known sensor model as stated above, i.e. a sensor reading of “nothing” is slightly more likely to get when the robot is close to a wall. Thus, even a “nothing” reading from the sensor should entail a proper prediction + update step.

Evaluate your implementation!

Hint 3: In terms of robot localisation it is often not relevant to know how often you are 100% correct with your estimate, but rather, how far "off" your estimate is from reality on average / how often. Measure the distance between true location and estimate by using the Manhattan distance (how many robot steps off), or the direct Euclidean distance (looks nicer, but would not help a robot that can only move straight too much).

Hint 4: Assume a grid size of preferably 8x8 (at least, however, 5x5) to base your evaluation on. If you use Java and an 8x8 grid, you should observe something like 30-35% of correct estimates rather quickly, roughly 100 steps should already get you there safely - note that this is only given as a reference for you, see hint 3 about evaluation of tracking systems. The average Manhattan distance (i.e. the number of “robot-steps” necessary to get from estimated to true position) should then be somewhere between 1.6 and 2.0. It is quite common to observe a checker-board pattern in the visualisation (see to the right, where the darker colours stand for higher, lighter for lower probabilities, grey is the highest one found, black is the true location and cyan marks the current sensor reading).



Hint 5: Code stubs and skeleton

JAVA - You can find a zip-file with a Java-based tool for visualisation, which now also contains implementations of the transition model and the sensor model and a stub for the core of the system - the ***DummyLocalizer***, that implements the ***EstimatorInterface***. Your solution should thus go into your own implementation (use the ***DummyLocalizer*** and add attributes as needed and implement the methods ***update*** and ***getCurrentProbDist***, and you can make use of the class stubs ***RobotSim*** and ***HMMFilter,*** or build your own class, but make sure to ***implement EstimatorInterface***) included in this archive, where you should also find a "user's guide" of the visualisation!

PYTHON - In this archive, you can find a Jupyter notebook, also this provides the state model as well as implementations for transition and sensor models, a stubb for a Localizer and some visualisation. Parts where your implementation for the robot simulation, filter algorithm and the updating loop should go are marked out accordingly. Note that the Python skeleton is new for 2021 - if something seems not to be corresponding to the description in this document or not working as intended, notify us! The notebook contains instructions to get the packages for the visualisation to work - there might be certain hiccups, do not hesitate to contact us, this should not keep you from working on the implementation.

If you experience difficulties with understanding the task, setting up the models, getting the implementation to work, etc, please, CONTACT us well BEFORE the deadline!

Elin: elin_anna.topp@cs.lth.se
Alexander: alexander.durr@cs.lth.se
Faseeh: faseeh.ahmad@cs.lth.se
Momina: momina.rizwan@cs.lth.se