Mini-project 2

Bookmark this page

Mini-project: Robot Localization

100/100 points (graded)

Update (October 26, 2016): *The deadline has been extended to Wednesday November 9, 2016 at 5pm Eastern Time, where importantly, due to daylight savings, this corresponds to UTC minus 5 hours rather than UTC minus 4 hours!*

Your pet robot is stuck on Mars somewhere on a grid of width 12 and height 8. You would like to figure out where your robot is over time so that you can rescue it.

CODE AND DATA

Let's first take a look at what you're getting yourself into. Download `robot.zip` and unzip it into a directory of your choice. You should see 5 files:

- `inference.py` — your code goes here; also shows how to generate HMM samples

Please don't modify:

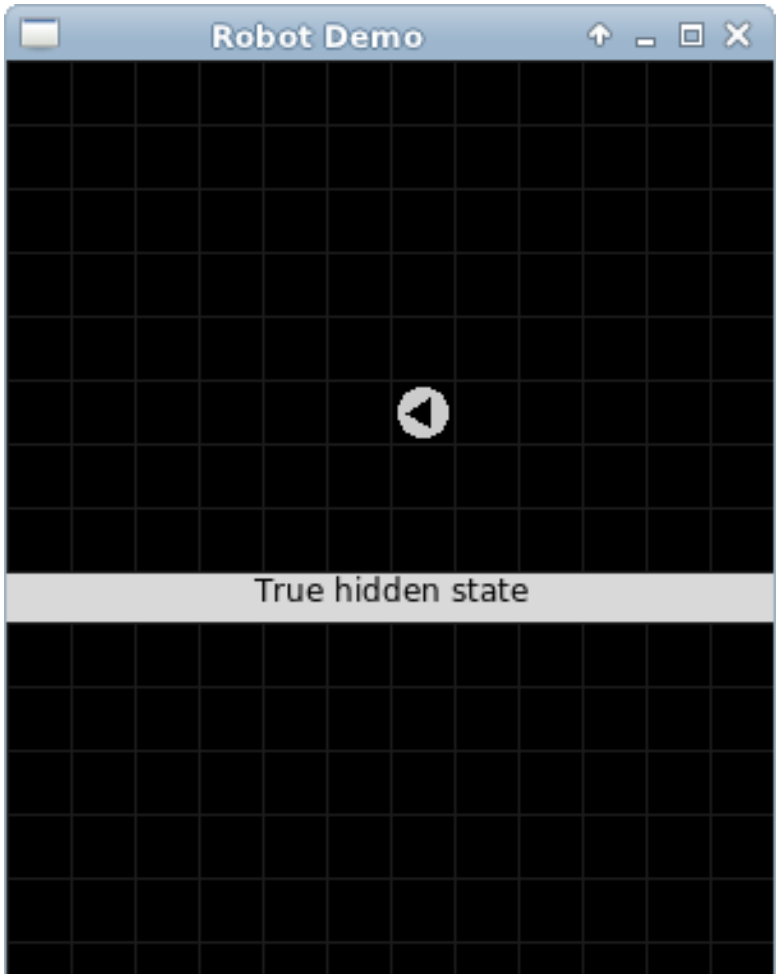
- `graphics.py` — graphics code; you need not read this
- `robot.py` — please read over this at the level of being able to call functions from it
- `test.txt` — data for parts (a), (c), and (d)

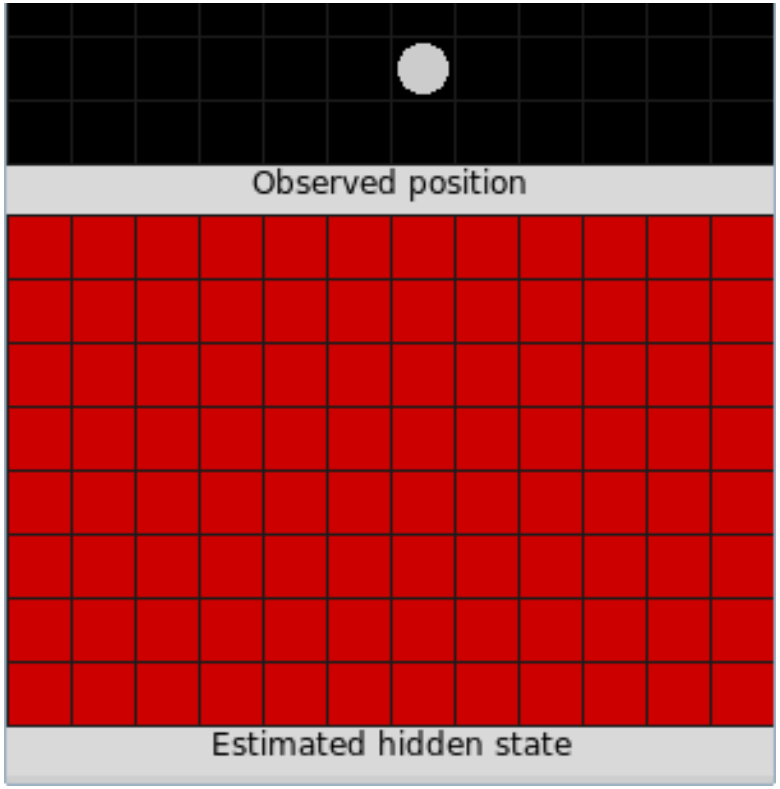
- `test_missing.txt` — data for parts (b) and (c)

You should be able to run the following and see your robot move around:

```
python inference.py
```

What you should see is an animated version of the following:





After you write in some inference algorithms, the bottom pane will automatically show something interesting rather than all red, signifying missing data. Also, the middle pane's robot lacks an arrow compared to the top pane's robot. The reason for the arrow being missing in the middle pane is that the actual robot state includes the robot's most recent action taken (represented by an arrow), which we don't get to observe so we'll be inferring these actions, such as moving left or right.

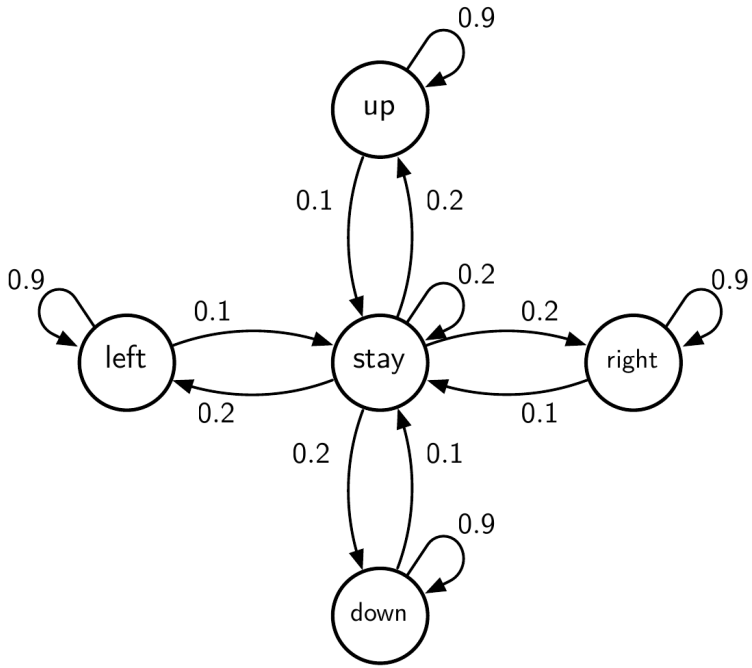
The staff solution for this lab runs in 2 seconds.

MODEL

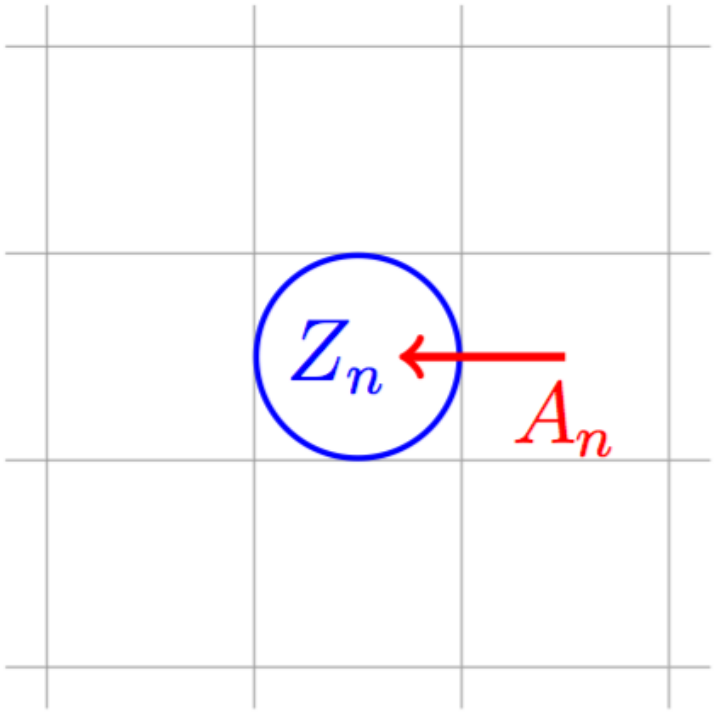
The robot's position at time i is given by random variable Z_i, which takes on a value in $\{0, 1, \ldots, 11\} \times \{0, 1, \ldots, 7\}$. For example, if $Z_2 = (5, 4)$, then this means that at time step 2, the robot is in column 5, row 4. Luckily, the robot is quite predictable. At each time step, it makes one of five actions: it stays put, goes left, goes up, goes right, or goes down. But the action it chooses depends on its previous action.

In particular, if its previous action was a movement, it moves in the same direction with probability 0.9 and stays put with probability 0.1. If its previous action was to stay put, it stays again (w.p. 0.2), or moves in any direction (each w.p. 0.2). For example, if the robot's previous action was ‘up', then with probability 0.1, the robot's next action will be ‘stay', and with probability 0.9, the robot's next action will be ‘up'.

We can visually represent these transitions with the following transition diagram (a transition diagram, while graphical and with probabilities is *not* the same as a probabilistic graphical model; each node in a probabilistic graphical model represents a random variable whereas in a transition diagram, each node is *not* a random variable but is instead an actual state that a random variable can take on, and the directed edges specify transition probabilities to get to other states):

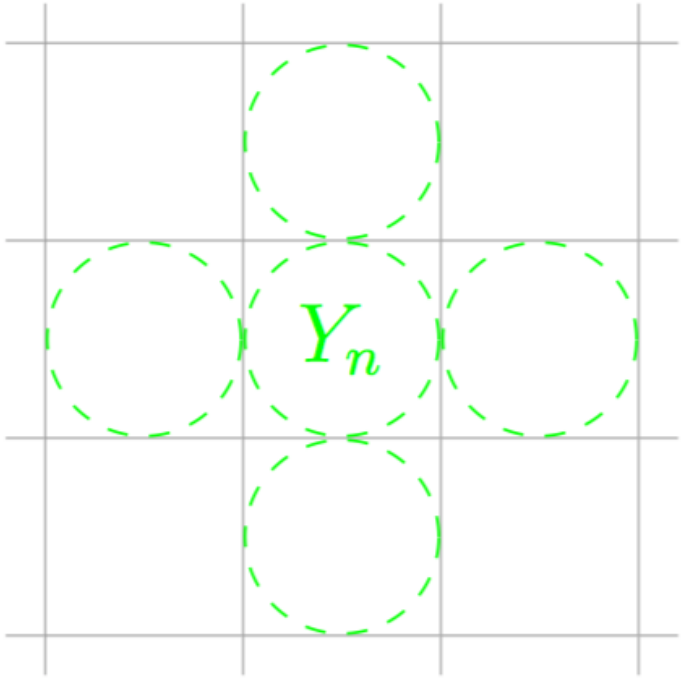


There's a catch: We need to treat the boundary of the grid differently. For instance, at the top of the grid, the robot can't go any higher. We'll renormalize the probabilities of its remaining actions at the top so that they sum to 1. Such boundary cases suggest that the transition probabilities depend on the robot's current location *and* its previous action. Thus, we model the robot's hidden state X_i at time i to consist of both its location Z_i and the action A_i it last took to reach Z_i, i.e., $X_i = (Z_i, A_i)$ as depicted below:



The state of the robot $X_i = (Z_i, A_i)$ where red shows action A_i and blue shows position Z_i.

Unfortunately, we will not have access to directly observing the robot's hidden state X_i. Instead, we have access to a noisy sensor that puts a uniform distribution on valid grid positions within 1 grid cell of the robot's current true position. Also, this noisy sensor only gives a guess as to the robot's current position and tells us nothing about what actions your robot has taken. In other words, at time i, we observe Y_i, which takes on a value in $\{0, 1, \ldots, 11\} \times \{0, 1, \ldots, 7\}$ as depicted below:



Distribution over observations Y_n, given the true state X_n. Each of these 5 possible observations is equally likely.

Lastly, we shall assume that initially the robot is in any of the grid locations with equal probability and always starts with its action set to 'stay'.

Tip: We have provided code in `robot.py` for generating the initial distribution, the transition probabilities given a current hidden state, and the observation probabilities given a current hidden state, so you need not reimplement these.

- **(a)** For every time step i, you want to compute a distribution which tells us where the robot was at time i. Implement the forward-backward algorithm to compute the marginal distribution $p_{X_i|Y_0,\ldots,Y_{n-1}}(\cdot|y_0,\ldots,y_{n-1})$ for each i. Specifically, fill out the function `forward_backward` in `inference.py`.

  Testing your code: Run:

  ```
  python inference.py --load=test.txt
  ```

  The marginal for time $i = 2$ should be

  $$p_{X_1|Y_0,\ldots,Y_{99}}(x_1 \mid y_0,\ldots,y_{99}) = \begin{cases} 0.7550047664442325 & \text{if } x = (8, 5, \text{`down'}) \\ 0.015252621544327934 & \text{if } x = (8, 5, \text{`stay'}) \\ 0.22974261201143953 & \text{if } x = (8, 6, \text{`down'}) \\ 0 & \text{otherwise} \end{cases}$$

  Tip for Getting Your Code to Run Fast: The forward-backward algorithm takes $\mathcal{O}(nk^2)$ computations, where n is the total number of time steps, and k is the number of possible hidden states. However, for this problem, there is actually additional structure such that it's possible to compute all the node marginals in roughly $\mathcal{O}(nk)$ computations—$\mathcal{O}(nk)$ for pre-processing and $\mathcal{O}(n)$ computations for the actual message passing. Why is this the case? How do you modify the forward-backward algorithm to achieve this faster performance?

- **(b)** Some of the observations were lost when they were transmitted from Mars back to Earth. Modify `forward_backward` so that it can handle missing observations, i.e., where at some time steps, there is no observation. In a list of observations, a missing observation is stored as `None` rather than the usual tuple `(x, y)`.

Note: We have not talked about what happens when there are missing observations and part of the point of this part is for you to figure it out! Think about what the graphical model looks like when certain time steps do not have observations.

Testing your code: After you have modified your forward-backward algorithm to handle missing observations, run:

```
python inference.py --load=test_missing.txt
```

The mode (i.e., the state with highest probability) of the marginal at time $i = 99$ should be `(3, 0, 'right')`, which should have probability 0.9. To print out the marginal at time $i = 99$, change the line `timestep = 2` to `timestep = 99` shortly after the code in `main()` that runs `forward_backward`.

The questions below depend on material from week 7.

- **(c)** Now, you want to know which *sequence* of states the robot was most likely to have visited. Implement the Viterbi algorithm for finding the MAP estimate of the entire sequence of hidden states given a sequence of observations. Specifically, fill out the function `Viterbi` in `inference.py`.

  After implementing the Viterbi algorithm, modify it so that it handles missing observations (`None`s in the observation vector).

  Testing your code: Run:

  ```
  python inference.py --load=test_missing.txt
  ```

  The last state of the MAP estimate should be `(3, 0, 'right')`.

- **(d)** Implement a modified version of the Viterbi algorithm that outputs the second-best solution to the MAP problem rather than the best solution by filling out the function `second_best` in `inference.py`.

This part is intentionally meant to be more challenging. Think about how to modify the Viterbi algorithm.

Clarification: If the MAP estimate is not unique (i.e., there are multiple sequences that all achieve the highest maximum posterior probability), then a second-best sequence should be one of the other equally good MAP estimates. We have chosen the test case below (and the test cases for the autograder) carefully so that the MAP estimate is actually unique, so the second-best sequence has a different posterior probability than that of the MAP estimate.

Testing your code: Run:

```
python inference.py --load=test.txt
```

The five time steps in which the MAP sequence and the second-best sequence differ should be time steps 50, 51, 52, 53, and 54.

THIS IS WHERE YOU PASTE YOUR CODE

Warning: If your code takes too long to run on the simple test cases that the autograder uses (we run your code through a number of short test sequences), then the autograder will *not* consider your submission a success, even if your code is correct! Please make sure your code runs in a reasonable amount of time. The autograder gives up after 20 seconds of running your code.

```
1 #!/usr/bin/env python
2 # inference.py
3 # Base code by George H. Chen (georgehc@mit.edu) -- updated 10/18/2016
4 import collections
```

Press ESC then TAB or click outside of the code editor to exit

Correct

Test results

Hide output

CORRECT

Test: forward_backward([(4, 3), (4, 2), (3, 2), (4, 0), (2, 0), (2, 0), (3, 2), (4, 2), (2, 3), (3, 5)])

Output:

```
[[[[3, 3, "stay"], 1.0]], [[[3, 2, "up"], 1.0]], [[[3, 1, "up"], 1.0]], [[[3, 0, "up"], 1.0]], [[[3, 0, "stay"], 1.0]], [[[3, 0, "stay"], 1.0]], [[[3, 1, "down"], 1.0]], [[[3, 2, "down"], 1.0]], [[[3, 3, "down"], 1.0]], [[[3, 4, "down"], 1.0]]]
```

Test: forward_backward([(5, 0), (3, 0), (3, 0), (2, 0), (0, 0), (0, 1), (0, 1), (1, 2), (0, 3), (0, 4)])

Output:

```
[[[[4, 0, "stay"], 1.0]], [[[3, 0, "left"], 1.0]], [[[2, 0, "left"], 1.0]],
[[[1, 0, "left"], 1.0]], [[[0, 0, "left"], 0.9997011920156017], [[1, 0,
"stay"], 0.0002988079843983499]], [[[0, 0, "stay"], 0.9997011920156017],
[[1, 1, "down"], 0.0002988079843983499]], [[[0, 1, "down"],
0.9997011920156016], [[1, 1, "stay"], 0.0002988079843983499]], [[[0, 2,
"down"], 0.9997011920156016], [[1, 2, "down"], 0.0002988079843983499]],
[[[0, 2, "stay"], 0.027018951135556794], [[0, 3, "down"],
0.972682240880045], [[1, 3, "down"], 0.0002988079843983499]], [[[0, 3,
"down"], 0.027018951135556805], [[0, 3, "stay"], 0.09726822408800449], [[0,
4, "down"], 0.8754140167920403], [[1, 4, "down"], 0.0002988079843983499]]]
```

Test: forward_backward([(6, 0), (6, 2), (7, 2), (7, 3), (7, 4), (7, 5), (6, 5), (5, 6), None, (7, 7)])

Output:

```
[[[[6, 0, "stay"], 0.8635394456289979], [[6, 1, "stay"],
0.13646055437100216]], [[[6, 1, "down"], 0.8635394456289979], [[6, 1,
"stay"], 0.12281449893390198], [[6, 2, "down"], 0.01364605543710022]], [[[6,
2, "down"], 0.9863539445628998], [[6, 2, "stay"], 0.013646055437100216]],
[[[6, 3, "down"], 1.0]], [[[6, 4, "down"], 1.0]], [[[6, 5, "down"], 1.0]],
[[[6, 5, "stay"], 0.4193548387096775], [[6, 6, "down"],
0.5806451612903225]], [[[6, 6, "down"], 0.4193548387096775], [[6, 6,
"stay"], 0.5806451612903226]], [[[6, 6, "stay"], 0.1612903225806452], [[6,
7, "down"], 0.8064516129032258], [[7, 6, "right"], 0.03225806451612903]],
[[[6, 7, "down"], 0.0896057347670251], [[6, 7, "stay"], 0.8064516129032258],
[[7, 6, "right"], 0.07168458781362008], [[7, 6, "stay"],
0.03225806451612904]]]
```

Test: forward_backward([(6, 5), (7, 4), (8, 4), (10, 4), (10, 5), None, (11, 5), (11, 5), (9, 4), None])

Output:

```
[[[[6, 4, "stay"], 1.0]], [[[7, 4, "right"], 1.0]], [[[8, 4, "right"], 1.0]], [[[9, 4, "right"], 1.0]], [[[10, 4, "right"], 1.0]], [[[10, 4, "stay"], 0.08924581457854344], [[11, 4, "right"], 0.9107541854214565]], [[[10, 5, "down"], 0.008289886985525085], [[11, 4, "right"], 0.08095592759301837], [[11, 4, "stay"], 0.9107541854214565]], [[[10, 5, "stay"], 0.008289886985525087], [[11, 4, "stay"], 0.9917101130144749]], [[[9, 5, "left"], 0.004144943492762544], [[10, 4, "left"], 0.9917101130144749], [[10, 4, "up"], 0.004144943492762544]], [[[8, 5, "left"], 0.003730449143486289], [[9, 4, "left"], 0.8925391017130273], [[9, 5, "stay"], 0.00041449434927625436], [[10, 3, "up"], 0.003730449143486289], [[10, 4, "stay"], 0.09958550565072373]]]
```

Test: Viterbi([(2, 0), (2, 0), (3, 0), (4, 0), (4, 0), (6, 0), (6, 1), (5, 0), (6, 0), (6, 2)])

Output:

```
[[1, 0, "stay"], [2, 0, "right"], [3, 0, "right"], [4, 0, "right"], [5, 0, "right"], [6, 0, "right"], [6, 0, "stay"], [6, 0, "stay"], [6, 1, "down"], [6, 2, "down"]]
```

Test: Viterbi([(1, 6), (4, 6), (4, 7), None, (5, 6), (6, 5), (6, 6), None, (5, 5), (4, 4)])

Output:

```
[[2, 6, "stay"], [3, 6, "right"], [4, 6, "right"], [5, 6, "right"], [6, 6, "right"], [6, 6, "stay"], [6, 5, "up"], [6, 5, "stay"], [5, 5, "left"], [4, 5, "left"]]
```

Test: second_best([(8, 2), (8, 1), (10, 0), (10, 0), (10, 1), (11, 0), (11, 0), (11, 1), (11, 2), (11, 2)])

Output:

```
[[9, 2, "stay"], [9, 1, "up"], [9, 0, "up"], [9, 0, "stay"], [10, 0, "right"], [11, 0, "right"], [11, 0, "stay"], [11, 0, "stay"], [11, 1, "down"], [11, 2, "down"]]
```

Test: second_best([(1, 4), (1, 5), (1, 5), (1, 6), (0, 7), (1, 7), (3, 7), (4, 7), (4, 7), (4, 7)])

Output:

```
[[1, 4, "stay"], [1, 5, "down"], [1, 6, "down"], [1, 7, "down"], [1, 7, "stay"], [1, 7, "stay"], [2, 7, "right"], [3, 7, "right"], [4, 7, "right"], [5, 7, "right"]]
```

Hide output

Submit You have used 13 of 100 attempts

✔ Correct (100/100 points)

Discussion

Topic: Mini-project 2 / Mini-project: Robot Localization

Show Discussion