Homework 4 - Car Tracking

Addapted from Stanford CS221 Fall 2019-2020

TA responsible: Xiaoqiang Lin

General Instructions

This (and every) assignment has a written part and a programming part.

The full assignment with our supporting code and scripts can be downloaded as car.zip.

a. This icon means a written answer is expected in `car.pdf`.

b. This icon means you should write code in `submission.py`.

All written answers must be **in order** and **clearly and correctly labeled** to receive credit.

You should modify the code in `submission.py` between

```
# BEGIN_YOUR_CODE
```

and

```
# END_YOUR_CODE
```

but you can add other helper functions outside this block if you want. Do not make changes to files other than `submission.py`.

Your code will be evaluated on two types of test cases, **basic** and **hidden**, which you can see in `grader.py`. Basic tests, which are fully provided to you, do not stress your code with large inputs or tricky corner cases. Hidden tests are more complex and do stress your code. The inputs of hidden tests are provided in `grader.py`, but the correct outputs are not. To run the tests, you will need to have `graderUtil.py` in the same directory as your code and `grader.py`. Then, you can run all the tests by typing

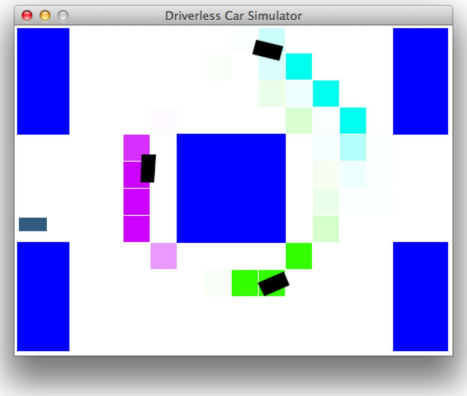
```
python grader.py
```

This will tell you only whether you passed the basic tests. On the hidden tests, the script will alert you if your code takes too long or crashes, but does not say whether you got the correct output. You can also run a single test (e.g., `3a-0-basic`) by typing

```
python grader.py 3a-0-basic
```

We strongly encourage you to read and understand the test cases, create your own test cases, and not just blindly run `grader.py`.

This assignment is a modified version of the Driverless Car assignment written by Chris Piech.



A study by the World Health Organization found that road accidents kill a shocking 1.24 million people a year worldwide. In response, there has been great interest in developing autonomous driving technology that can drive with calculated precision and reduce this death toll. Building an autonomous driving system is an incredibly complex endeavor. In this assignment, you will focus on the sensing system, which allows us to track other cars based on noisy sensor readings.

Getting started. You will be running two files in this assignment - grader.py and drive.py. grader.py runs on Python 3 as usual, but **drive.py runs on Python 2**. drive.py is not used for any grading purposes, it's just there to visualize the code you will be writing and help you gain an appreciation for how different approaches result in different behaviors (and to have fun!). Let's start by trying to drive manually. *Note again that you need to run drive.py using Python2.*

```
python drive.py -l lombard -i none
```

You can steer by either using the arrow keys or 'w', 'a', and 'd'. The up key and 'w' accelerates your car forward, the left key and 'a' turns the steering wheel to the left, and the right key and 'd' turns the steering wheel to the right. Note that you cannot reverse the car or turn in place. Quit by pressing 'q'. Your goal is to drive from the start to finish (the green box) without getting in an accident. How well can you do on the windy Lombard street without knowing the location of other cars? Don't worry if you're not very good; the teaching staff were only able to get to the finish line 4/10 times. An accident rate of 60% is pretty abysmal, which is why we're going to use AI to do this.

Flags for `python drive.py`:

- `-a`: Enable autonomous driving (as opposed to manual).
- `-i <inference method>`: Use `none`, `exactInference`, `particleFilter` to (approximately) compute the belief distributions over the locations of the other cars.
- `-l <map>`: Use this map (e.g. `small` or `lombard`). Defaults to `small`.
- `-d`: Debug by showing all the cars on the map.
- `-p`: All other cars remain parked (so that they don't move).

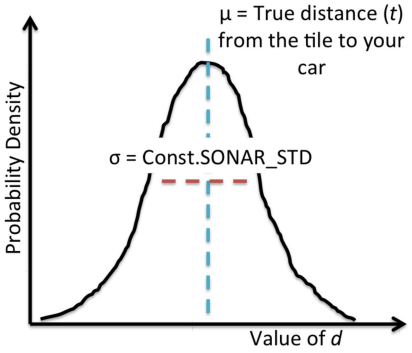
Modeling car locations. We assume that the world is a two-dimensional rectangular grid on which your car and K other cars reside. At each time step t, your car gets a noisy estimate of the distance to each of the cars. As a simplifying assumption, we assume that each of the K other cars

moves independently and that the noise in sensor readings for each car is also independent. Therefore, in the following, we will reason about each car independently (notationally, we will assume there is just one other car).

At each time step t, let $C_t \in \mathbb{R}^2$ be a pair of coordinates representing the *actual* location of the single other car (which is unobserved). We assume there is a local conditional distribution $p(c_t \mid c_{t-1})$ which governs the car's movement. Let $a_t \in \mathbb{R}^2$ be your car's position, which you observe and also control. To minimize costs, we use a simple sensing system based on a microphone. The microphone provides us with D_t, which is a Gaussian random variable with mean equal to the true distance between your car and the other car and variance σ^2 (in the code, σ is `Const.SONAR_STD`, which is about two-thirds the length of a car). In symbols,

$$D_t \sim \mathcal{N}(\|a_t - C_t\|, \sigma^2).$$

For example, if your car is at $a_t = (1, 3)$ and the other car is at $C_t = (4, 7)$, then the actual distance is 5 and D_t might be 4.6 or 5.2, etc. Use `util.pdf(mean, std, value)` to compute the probability density function (PDF) of a Gaussian with given mean `mean` and standard deviation `std`, evaluated at `value`. Note that evaluating a PDF at a certain value does not return a probability -- densities can exceed 1 -- but for the purposes of this assignment, you can get away with treating it like a probability. The Gaussian probability density function for the noisy distance observation D_t, which is centered around your distance to the car $\mu = \|a_t - C_t\|$, is shown in the following figure:



Your job is to implement a car tracker that (approximately) computes the posterior distribution $\mathbb{P}(C_t \mid D_1 = d_1, \ldots, D_t = d_t)$ (your beliefs of where the other car is) and update it for each $t = 1, 2, \ldots$. We will take care of using this information to actually drive the car (i.e., set a_t to avoid a collision with c_t), so you don't have to worry about that part.

To simplify things, we will discretize the world into **tiles** represented by `(row, col)` pairs, where `0 <= row < numRows` and `0 <= col < numCols`. For each tile, we store a probability representing our belief that there's a car on that tile. The values can be accessed by: `self.belief.getProb(row, col)`. To convert from a tile to a location, use `util.rowToY(row)` and `util.colToX(col)`.

Here's an overview of the assignment components:

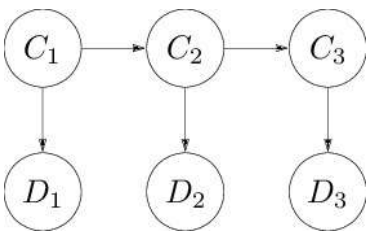
- Problem 1 (written) will give you some practice with probabilistic inference on a simple Bayesian network.
- In Problems 2 and 3 (code), you will implement `ExactInference`, which computes a full probability distribution of another car's location over tiles `(row, col)`.
- In Problem 4 (code), you will implement `ParticleFilter`, which works with particle-based representation of this same distribution.

A few important notes before we get started:

- Past experience suggests that this will be one of the most conceptually challenging assignments of the quarter for many students. Please start early, especially if you're low on late days!
- We strongly recommend that you attend/watch the lectures on Bayesian networks and HMMs before getting started, and keep the slides handy for reference while you're working.
- The code portions of this assignment are short and straightforward -- no more than about 30 lines in total -- but only if your understanding of the probability concepts is clear! (If not, see the previous point.)
- As a notational reminder: we use the lowercase expressions $p(x)$ or $p(x|y)$ for local conditional probability distributions, which are defined by the Bayesian network. We use the uppercase expressions $\mathbb{P}(X = x)$ or $\mathbb{P}(X = x|Y = y)$ for joint and posterior probability distributions, which are not pre-defined in the Bayesian network but can be computed by probabilistic inference. Please review the lecture slides for more details.
- As mentioned at the start of the assignment, remember to run `drive.py` with Python 2. Note that drive.py isn't used for grading purposes.

Problem 1: Bayesian network basics

First, let us look at a simplified version of the car tracking problem. For this problem only, let $C_t \in \{0, 1\}$ be the actual location of the car we wish to observe at time step $t \in \{1, 2, 3\}$. Let $D_t \in \{0, 1\}$ be a sensor reading for the location of that car measured at time t. Here's what the Bayesian network (it's an HMM, in fact) looks like:



The distribution over the initial car distribution is uniform; that is, for each value $c_1 \in \{0, 1\}$:

$$p(c_1) = 0.5.$$

The following local conditional distribution governs the movement of the car (with probability ϵ, the car moves). For each $t \in \{2, 3\}$:

$$p(c_t \mid c_{t-1}) = \begin{cases} \epsilon & \text{if } c_t \neq c_{t-1} \\ 1 - \epsilon & \text{if } c_t = c_{t-1}. \end{cases}$$

The following local conditional distribution governs the noise in the sensor reading (with probability η, the sensor reports the *wrong* position). For each $t \in \{1, 2, 3\}$:

$$p(d_t \mid c_t) = \begin{cases} \eta & \text{if } d_t \neq c_t \\ 1 - \eta & \text{if } d_t = c_t. \end{cases}$$

Below, you will be asked to find the posterior distribution for the car's position at the second time step (C_2) given different sensor readings.

Important: For the following computations, try to follow the general strategy described in lecture (marginalize non-ancestral variables, condition, and perform variable elimination). Try to delay normalization until the very end. You'll get more insight than trying to chug through lots of equations.

a. Suppose we have a sensor reading for the second timestep, $D_2 = 0$. Compute the posterior distribution $\mathbb{P}(C_2 = 1 \mid D_2 = 0)$. We encourage you to draw out the (factor) graph.

b. Suppose a time step has elapsed and we got another sensor reading, $D_3 = 1$, but we are still interested in C_2. Compute the posterior distribution $\mathbb{P}(C_2 = 1 \mid D_2 = 0, D_3 = 1)$. The resulting expression might be moderately complex. We encourage you to draw out the (factor) graph.

c. Suppose $\epsilon = 0.1$ and $\eta = 0.2$.

i. Compute and compare the probabilities $\mathbb{P}(C_2 = 1 \mid D_2 = 0)$ and $\mathbb{P}(C_2 = 1 \mid D_2 = 0, D_3 = 1)$. Give numbers, round your answer to 4 significant digits.

ii. How did adding the second sensor reading $D_3 = 1$ change the result? Explain your intuition for why this change makes sense in terms of the car positions and associated sensor observations.

iii. What would you have to set ϵ to be while keeping $\eta = 0.2$ so that $\mathbb{P}(C_2 = 1 \mid D_2 = 0) = \mathbb{P}(C_2 = 1 \mid D_2 = 0, D_3 = 1)$? Explain your intuition in terms of the car positions with respect to the observations.

Problem 2: Emission probabilities

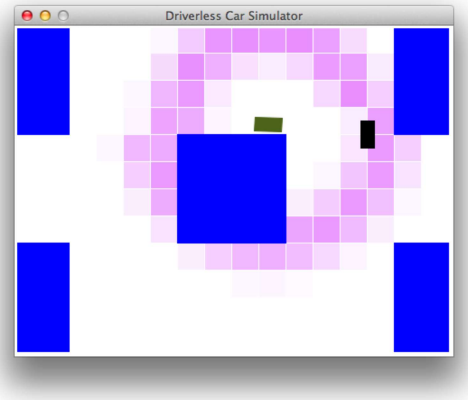
In this problem, we assume that the other car is stationary (e.g., $C_t = C_{t-1}$ for all time steps t). You will implement a function `observe` that upon observing a new distance measurement $D_t = d_t$ updates the current posterior probability from

$$\mathbb{P}(C_t \mid D_1 = d_1, \ldots, D_{t-1} = d_{t-1})$$

to

$$\mathbb{P}(C_t \mid D_1 = d_1, \ldots, D_t = d_t) \propto \mathbb{P}(C_t \mid D_1 = d_1, \ldots, D_{t-1} = d_{t-1}) p(d_t \mid c_t),$$

where we have multiplied in the emission probabilities $p(d_t \mid c_t)$ described earlier under "Modeling car locations". The current posterior probability is stored as `self.belief` in `ExactInference`.



a. Fill in the `observe` method in the `ExactInference` class of `submission.py`. This method should modify `self.belief` in place to update the posterior probability of each tile given the observed noisy distance to the other car. After you're done, you should be able to find the stationary car by driving around it (using the flag `-p` means cars don't move):

Notes:

- You can start driving with exact inference now. Remember to use Python 2 when running drive.py.

```
python drive.py -a -p -d -k 1 -i exactInference
```

  You can also turn off `-a` to drive manually.

- It's generally best to run `drive.py` on your local machine, but if you do decide to run it on cardinal/rice instead, please ssh into the farmshare machines with either the **-X** or **-Y** option in order to get the graphical interface; otherwise, you will get some display error message. Note: do expect this graphical interface to be a bit slow... `drive.py` is not used for grading, but is just there for you to visualize and enjoy the game!
- Read through the code in util.py for the `Belief` class before you get started... you'll need to use this class for several of the code tasks in this assignment.
- Remember to normalize the posterior probability after you update it. (There's a useful function for this in util.py).
- On the small map, the autonomous driver will sometimes drive in circles around the middle block before heading for the target area. In general, don't worry too much about the precise path the car takes. Instead, focus on whether your car tracker correctly infers the location of other cars.
- Don't worry if your car crashes once in a while! Accidents do happen, whether you are human or AI. However, even if there was an accident, your driver should have been aware that there was a high probability that another car was in the area.

Problem 3: Transition probabilities

Now, let's consider the case where the other car is moving according to transition probabilities $p(c_{t+1} | c_t)$. We have provided the transition probabilities for you in `self.transProb`. Specifically, `self.transProb[(oldTile, newTile)]` is the probability of the other car being in `newTile` at time step $t+1$ given that it was in `oldTile` at time step t.

In this part, you will implement a function `elapseTime` that updates the posterior probability about the location of the car at a **current** time t

$$\mathbb{P}(C_t = c_t \mid D_1 = d_1, \ldots, D_t = d_t)$$

to the **next** time step $t+1$ conditioned on the same evidence, via the recurrence:

$$\mathbb{P}(C_{t+1} = c_{t+1} | D_1 = d_1, \ldots, D_t = d_t) \propto \sum_{c_t} \mathbb{P}(C_t = c_t \mid D_1 = d_1, \ldots, D_t = d_t) p(c_{t+1} | c_t).$$

Again, the posterior probability is stored as `self.belief` in `ExactInference`.

a. Finish `ExactInference` by implementing the `elapseTime` method. When you are all done, you should be able to track a moving car well enough to drive autonomously by running the following. Remember to run drive.py using Python 2.

```
python drive.py -a -d -k 1 -i exactInference
```

Notes:

- You can also drive autonomously in the presence of more than one car:

  ```
  python drive.py -a -d -k 3 -i exactInference
  ```

- You can also drive down Lombard:

  ```
  python drive.py -a -d -k 3 -i exactInference -l lombard
  ```

  Remember to use Python 2 for running drive.py. On Lombard, the autonomous driver may attempt to drive up and down the street before heading towards the target area. Again, focus on the car tracking component, instead of the actual driving.

Problem 4: Particle filtering

Though exact inference works well for the small maps, it wastes a lot of effort computing probabilities for *every available tile*, even for tiles that are unlikely to have a car on them. We can solve this problem using a particle filter. Updates to the particle filter have complexity that's linear in the number of particles, rather than linear in the number of tiles.

For a great conceptual explanation of how particle filtering works, check out this video on using particle filtering to estimate an airplane's altitude.

In this problem, you'll implement two short but important methods for the `ParticleFilter` class in `submission.py`. When you're finished, your code should be able to track cars nearly as effectively as it does with exact inference.

a. Some of the code has been provided for you. For example, the particles have already been initialized randomly. You need to fill in the `observe` and `elapseTime` functions. These should modify `self.particles`, which is a map from tiles `(row, col)` to the number of particles existing at that tile, and `self.belief`, which needs to be updated each time you re-sample the particle locations.

You should use the same transition probabilities as in exact inference. The belief distribution generated by a particle filter is expected to look noisier compared to the one obtained by exact inference. Remember to run drive.py with Python 2.

```
python drive.py -a -i particleFilter -l lombard
```

To debug, you might want to start with the parked car flag (`-p`) and the display car flag (`-d`).

Submission

Submit a zip file consisting of both car.pdf and submission.py via E-learning platform before mid-night of Jan 5th, 2020.

Name the zip file as hw4-your-sid.zip.

For any question about this assignment, contact Xiaoqiang Lin:16307100046@fudan.edu.cn for more information.
