Assignment 2. Car Tracking

Gary Geunbae Lee
CSED342 - Artificial Intelligence

Contact: TA Baikjin Jung (bjjung@postech.ac.kr)

General Instructions

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

You should write both types of answers in `submission.py` between

```
# BEGIN_YOUR_ANSWER
```

and

```
# END_YOUR_ANSWER
```

✎This icon means a written answer is expected. Some of these problems are multiple choice questions that impose negative scores if the answers are incorrect. So, don't write answers unless you are confident.

⌨This icon means you should write code. you can add other helper functions outside the answer 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 all the tests, type

```
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`.

Introduction

This assignment is a modified version of the Driverless Car assignment written by Chris Piech. Your code will exploit a random number generator, but its behavior is different depending on your Python version. Therefore, we recommend you to use the same Python version and distribution with those of TA.

A study by the World Health Organisation 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 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. Let's start by trying to drive manually:

```
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 crooked Lombard street without knowing the location of other cars? Don't worry if you aren't very good; the staff was only able to get to the finish line 4/10 times. This 60% accident rate is pretty abysmal, which is why we're going to build an AI to do this.

Flags for `python drive.py`:

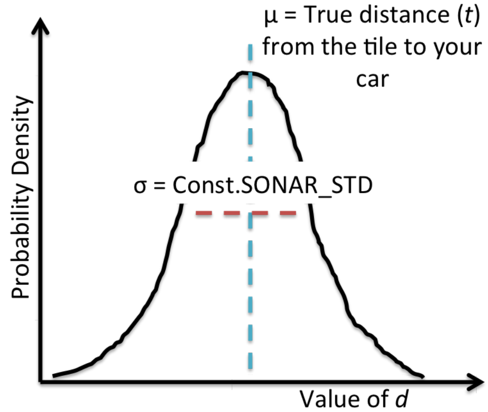
- `-a`: Enable autonomous driving (as opposed to manual).
- `-i <inference method>`: Use `none`, `exactInference`, `particleFilter` to (approximately) compute the belief distributions.
- `-l <map>`: Use this map (e.g. `small` or `lombard`). Defaults to `small`.
- `-d`: Debug by showing all the cars on 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).

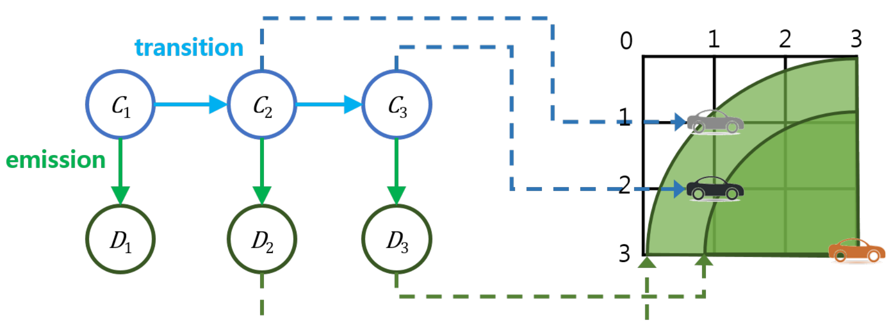
At each time step t, let $C_t \in \mathbb{R}^2$ be a pair of coordinates representing the actual location of the single other car (which is unobserved). We assume there is a local conditional distribution $p(c_t \mid c_{t-1})$ which governs the car's movement. Let $a_t \in \mathbb{R}^2$ be your car's 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 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,

$$D_t \sim \mathcal{N}(\|a_t - C_t\|, \sigma^2). \tag{1}$$

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 and standard deviation, evaluated at `value`. Note that the PDF 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 $\mu = \|a_t - C_t\|$:



The figure below shows another example where one black car and our (orange) car are located in the 2D grid and the sensor estimates the distances of the black car from our car. Note that we can access the information of distances (D_t) measured by the sensor, but we're not provided with the exact locations (C_t) of the black car.



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 actual drive the car (i.e., set a_t as to avoid collision with c_t), so you don't have to worry about that part.

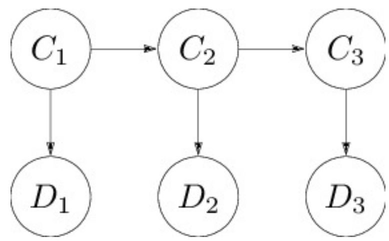
To simplify things, we will discretize the world into **tiles** represented by `(row, col)` pairs, where `0` $\leq$ `row` $<$ `numRows` and `0` $\leq$ `col` $<$ `numCols`. For each tile, we store a probability distribution whose values can be accessed by `self.belief.getProb(row, col)`. To convert from a tile to a location, use `util.rowToY(row)` and `util.colToX(col)`.

Problem 1 will be a warmup. In Problems 2 and 3, you will implement `ExactInference`, which computes a full distribution over tiles `(row, col)`. In Problem 4, you will implement `ParticleFilter`, which works with particle-based represention of this distribution.

Note: as a notational reminder, the lower case p(x) is the local distribution defined by the user. On the other hand, the quantity $\mathbb{P}(X = x)$ is not defined, but follows from probabilistic inference. Please review lecture slides for more details.

Problem 1: Warmup

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 position $c_1 \in \{0, 1\}$ is defined as:

$$p(c_1) = \begin{cases} \delta & \text{if } c_1 = 0 \\ 1 - \delta & \text{if } c_1 = 1. \end{cases}.$$

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.

Problem 1a [4 points]

Suppose we have a sensor reading for the second timestep, $D_2 = d_2$. Compute the posterior distribution $\mathbb{P}(C_2 = c_2 \mid D_2 = d_2)$ by implementing `get_conditional_prob1`.

Problem 1b [4 points]

Suppose a time step has elapsed and we got another sensor reading, $D_3 = d_3$, but we are still interested in C_2. Compute the posterior distribution $\mathbb{P}(C_2 = c_2 \mid D_2 = d_2, D_3 = d_3)$ by implementing `get_conditional_prob2`.

Problem 1c [2 points] ✎

If we set ϵ to a specific value, we can see $\mathbb{P}(C_2 = c_2 \mid D_2 = d_2) = \mathbb{P}(C_2 = c_2 \mid D_2 = d_2, D_3 = d_3)$. What's the value? `get_epsilon` should return the value. (Think about the reason.)

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

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

to

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

where we have multiplied in the emission probabilities $p(d_t \mid c_t)$ described earlier. The current posterior probability is stored as `self.belief` in `ExactInference`, which you should update `self.belief` in place.

Problem 2a [4 points]

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

Notes:

- You can start driving with exact inference now.
  `python drive.py -a -p -d -k 1 -i exactInference`
  You can also turn off `-a` to drive manually.

- Remember to normalize the updated posterior probability (see useful functions provided in utils.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 driving the car. Instead, focus on if 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 $p(c_{t+1} \mid c_t)$. 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

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

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

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

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

Problem 3a [4 points]

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:

```
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
  ```

  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 cars being on unlikely tiles. We can solve this problem using a particle filter which has complexity linear in the number of particles rather than linear in the number of tiles. Implement all necessary methods for the `ParticleFilter` class in `submission.py`. When complete, you should be able to track cars nearly as effectively as with exact inference.

Problem 4a [6 points] ⌨

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 times that particle occurs, and `self.belief`, which needs to be updated after you resample the particles.

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.

```
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`).