Assignment 7. Car Tracking

Hwanjo Yu
CSED342 - Artificial Intelligence

Contact: TA Jinhwan Nam (njh18@postech.ac.kr)
Deadline: 26th May 2024 at 2:00 pm

This assignment has been developed in **Python 3.8**, so please use **Python 3.8** to implement your code. we recommend using **Conda environment**.[1]

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

```
# BEGIN_YOUR_ANSWER
```

and

```
# END_YOUR_ANSWER
```

You can add other helper functions outside the answer block if you want, but do not import other libraries and 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, 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. The script will alert you if your code takes too long or crashes on the hidden tests, but does not say whether you got the correct output. You can also run a single test (e.g., `2a-0-basic`) by typing

```
python grader.py 2a-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`.

[1]https://docs.conda.io/projects/conda/en/stable/user-guide/install/index.html

Introduction

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

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 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', and quit `drive.py` by using 'q'. Note that you cannot reverse the car or turn in place. 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 the all cars on the map.
- `-p`: All other cars remain parked (so that they don't move).

Modeling car locations

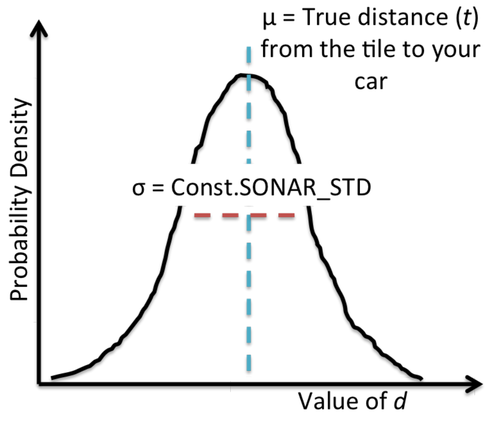
We assume that the world is a two-dimensional rectangular grid on which your car and K other cars reside. At each time step t, your car gets a noisy estimate of the distance to each of the cars. As a simplifying assumption, we assume that each of the K other cars moves independently and that the noise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

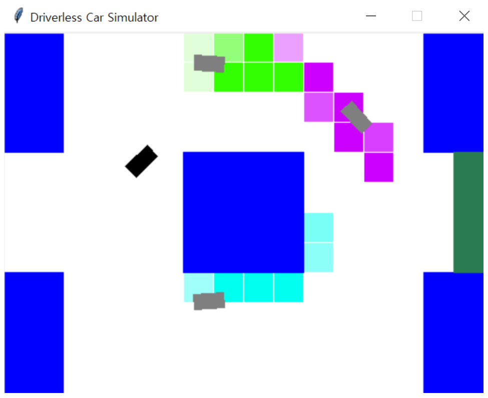
$p(c_t \mid c_{t-1})$ which governs the car's movement. Let $a_t \in \mathbb{R}^2$ be your car's position, which you observe and also control. To minimize costs, we use a simple sensing system based on a microphone. The microphone provides us with D_t, which is a Gaussian random variable with mean equal 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 a black car (ours) and gray cars are located in the 2D grid and the sensor estimates the distances of the black car from the other cars. 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 gray cars.



Your job is to implement a car tracker that (approximately) computes the posterior distribution $\mathbb{P}(C_t \mid D_1 = d_1, \dots, D_t = d_t)$ (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 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)`.

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

Note: as a reminder of notation, 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 the lecture slides for more details.

Problem 1: 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 1a [7 points]

Before jumping into the code, please read `util.py` to find useful functions.

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 by executing the following command:

  ```
  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 `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 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 2: 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 `ExactInference.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 2a [7 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 by executing:

```
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 3: 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 3a [18 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 of 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.

You can execute the following command to check the behavior of your code:

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

Notes:

- For debugging, you may use the park car flag (`-p`) and the display car flag (`-d`).
- If you have implemented `ParticleFilter` correctly, `self.belief` should never appear in your code. Also, `self.updateBelief` should never be called, except the 187th line of distributed `submission.py`, which is a part of `ParticleFilter.observe`. **0 points will be given** if the above conditions are violated even if the grader gives you a full point.
- In the implementation of random process, **you should use** `util.weightedRandomChoice` in both `observe` and `elapseTime`. The method `util.weightedRandomChoice` calls `random.uniform`, which is affected by a random seed and how many time it is called. Therefore, please read the following and write your code carefully.
  - In `observe`, `util.weightedRandomChoice` is used in a part that re-samples the particles, which should be written as a following form:

    ```
    for _ in range(self.NUM_PARTICLES): <your code>
    ```

  - In `elapseTime`, `util.weightedRandomChoice` is used in a part that finds the distribution of particles on the next time step, which should be written as a following form:

    ```
    for particle in self.particles: <your code>
    ```

  You may use different names for variables, but please follow the given structure.