XCS221 Assignment 6 — Car Tracking

Due Sunday, June 26 at 11:59pm PT.

Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at http://xcs221-scpd.slack.com/
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

Submission Instructions

Written Submission: Some questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Type responses only in `submission.tex`.
- Submit the compiled PDF, **not** `submission.tex`.
- Use the commented instructions within the `Makefile` and `README.md` to get started.

Coding Submission: Some questions in this assignment require a coding response. For these questions, you should submit only the `src/submission.py` file in the online student portal. For further details, see Writing Code and Running the Autograder below.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. More information regarding the Stanford honor code can be foudn at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

Writing Code and Running the Autograder

All your code should be entered into `src/submission.py`. When editing `src/submission.py`, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files other than `src/submission.py`.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These tests are provided to make sure that your inputs and outputs are on the right track, and that the hidden evaluation tests will be able to execute.

- `hidden`: These unit tests are the evaluated elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests. These evaluative hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

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

Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

Test Cases

The autograder is a thin wrapper over the python `unittest` framework. It can be run either locally (on your computer) or remotely (on SCPD servers). The following description demonstrates what test results will look like for both local and remote execution. For the sake of example, we will consider two generic tests: `1a-0-basic` and `1a-1-hidden`.

Local Execution - Hidden Tests

All hidden tests rely on files that are not provided to students. Therefore, the tests can only be run remotely. When a hidden test like `1a-1-hidden` is executed locally, it will produce the following result:

```
----- START 1a-1-hidden:  Test multiple instances of the same word in a sentence.
----- END 1a-1-hidden [took 0:00:00.011989 (max allowed 1 seconds), ???/3 points] (hidden test ungraded)
```

Local Execution - Basic Tests

When a basic test like `1a-0-basic` passes locally, the autograder will indicate success:

```
----- START 1a-0-basic:  Basic test case.
----- END 1a-0-basic [took 0:00:00.000062 (max allowed 1 seconds), 2/2 points]
```

When a basic test like `1a-0-basic` fails locally, the error is printed to the terminal, along with a stack trace indicating where the error occurred:

```
----- START 1a-0-basic:  Basic test case.
<class 'AssertionError'>
{'a': 2, 'b': 1} != None
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
----- END 1a-0-basic [took 0:00:00.003809 (max allowed 1 seconds), 0/2 points]
```


Remote Execution

Basic and hidden tests are treated the same by the remote autograder. Here are screenshots of failed basic and hidden tests. Notice that the same information (error and stack trace) is provided as the in local autograder, now for both basic and hidden tests.

1a-0-basic) Basic test case. (0.0/2.0)

```
<class 'AssertionError'>:  {'a': 2, 'b': 1} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

Just like in the local autograder, this error caused the test to fail.

Just like in the local autograder, start your debugging in line 23 of grader.py.

1a-1-hidden) Test multiple instances of the same word in a sentence. (0.0/3.0)

```
<class 'AssertionError'>:  {'a': 23, 'ab': 22, 'aa': 24, 'c': 16, 'b': 15} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 31, in test_1
    self.compare_with_solution_or_wait(submission, 'extractWordFeatures', lambda f: f(sentence))
  File "/autograder/source/graderUtil.py", line 183, in compare_with_solution_or_wait
    self.assertEqual(ans1, ans2)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

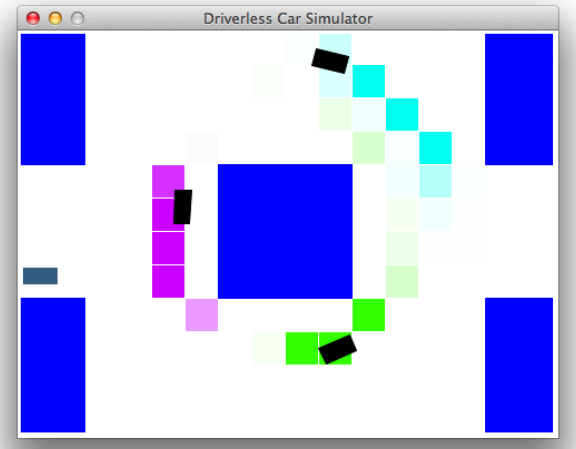
This error caused the test to fail.

Start your debugging in line 31 of grader.py.

Finally, here is what it looks like when basic and hidden tests pass in the remote autograder.

1a-0-basic) Basic test case. (2.0/2.0)

1a-1-hidden) Test multiple instances of the same word in a sentence. (3.0/3.0)



Introduction

This assignment is a modified version of the Driverless Car assignment written by Dr. 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`. The `drive.py` file 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!). 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 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 you'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).

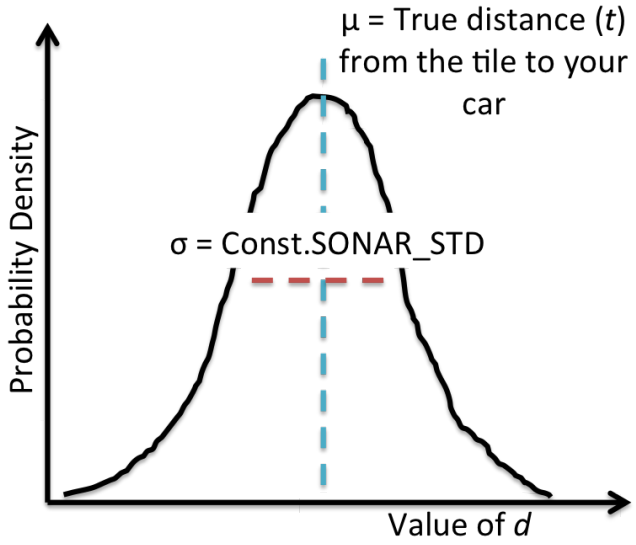
Modeling car locations.

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, 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, reason about each car independently (notationally, 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). 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, 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, discretize the world into **tiles** represented by `(row, col)` pairs, where `0 <= row < numRows` and `0 <= col < numCols`. For each tile, there is a stored probability representing the 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.

- Problem 5 (written) gives you a chance to extend your probability analyses to a slightly more realistic scenario where there are multiple other cars and you can't automatically distinguish between them.

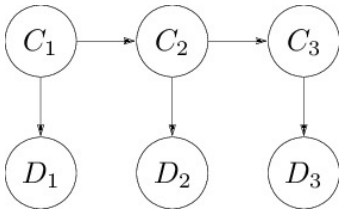
A few important notes before you get started:

- Past experience suggests that this will be one of the most conceptually challenging assignments for many students. Please start early!
- You should 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.

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) **[2 points (Written)]** 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) **[3 points (Written)]** 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) **[3 points (Written)]** 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.

2. Emission Probabilities

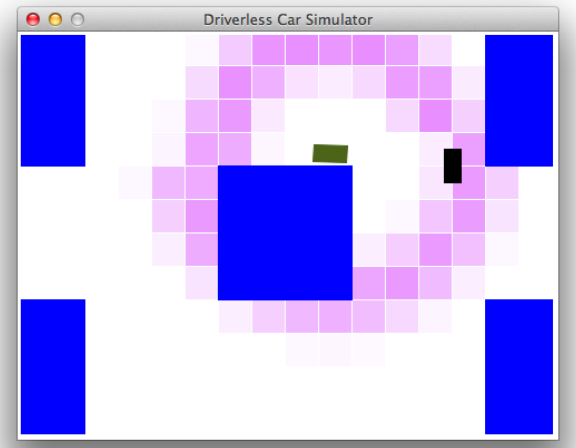
In this problem, we assume that the other car is stationary (e.g., $C_t = C_{t-1}$ for all time steps t). You will implement a function `observe` that upon observing a new distance measurement $D_t = d_t$ updates the current posterior probability from

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

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) **[7 points (Coding)]** 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.*

```
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 remote machine instead, please ssh with either the `-X` or `-Y` option in order to get the graphical interface; otherwise, you will get some display error message. Note: in this case, 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.*

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

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

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

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

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

(a) **[11 points (Coding)]** 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:

```
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.

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) **[20 points (Coding)]** 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.

```
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`).

5. **Which Car is it?**

So far, we have assumed that we have a distinct noisy distance reading for each car, but in reality, our microphone would just pick up an undistinguished set of these signals, and we wouldn't know which distance reading corresponds to which car. First, let's extend the notation from before: let $C_{ti} \in \mathbb{R}^2$ be the location of the i-th car at the time step t, for $i = 1, \ldots, K$ and $t = 1, \ldots, T$. Recall that all the cars move independently according to the transition dynamics as before.

Let $D_{ti} \in \mathbb{R}$ be the noisy distance measurement of the i-th car at time step t, which is now not directly observed. Instead, we observe the **set** of distances $D_t = \{D_{t1}, \ldots, D_{tK}\}$. (For simplicity, we'll assume that all distances are distinct values.) Alternatively, you can think of $E_t = (E_{t1}, \ldots, E_{tK})$ as a list which is a uniformly random permutation of the noisy distances $(D_{t1}, \ldots, D_{tK})$. For example, suppose $K = 2$ and $T = 2$. Before, we might have gotten distance readings of 1 and 2 for the first car and 3 and 4 for the second car at time steps 1 and 2, respectively. Now, our sensor readings would be permutations of $\{1, 3\}$ (at time step 1) and $\{2, 4\}$ (at time step 2). Thus, even if we knew the second car was distance 3 away at time $t = 1$, we wouldn't know if it moved further away (to distance 4) or closer (to distance 2) at time $t = 2$.

(a) **[4 points (Written)]** Suppose we have $K = 2$ cars and one time step $T = 1$. Write an expression for the conditional distribution $\mathbb{P}(C_{11}, C_{12} \mid E_1 = e_1)$ as a function of the PDF of a Gaussian $p_{\mathcal{N}}(v; \mu, \sigma^2)$ and the prior probability $p(c_{11})$ and $p(c_{12})$ over car locations. Your final answer should not contain variables d_{11}, d_{12}.

Remember that $p_{\mathcal{N}}(v; \mu, \sigma^2)$ is the probability of a random variable, v, in a Gaussian distribution with mean μ and standard deviation σ.

Hint: for $K = 1$, the answer would be

$$\mathbb{P}(C_{11} = c_{11} \mid E_1 = e_1) \propto p(c_{11}) p_{\mathcal{N}}(e_{11}; \|a_1 - c_{11}\|, \sigma^2).$$

where a_t is the position of the car at time t. You might find it useful to draw the Bayesian network and think about the distribution of E_t given $D_{t1}, \ldots, D_{tK}$.

(b) **[3 points (Written)]** Assuming the prior $p(c_{1i})$ of where the cars start out is the same for all i (i.e. for all K cars), show that the number of assignments for all K cars $(c_{11}, \ldots, c_{1K})$ that obtain the maximum value of $\mathbb{P}(C_{11} = c_{11}, \ldots, C_{1K} = c_{1K} \mid E_1 = e_1)$ is at least $K!$.

You can also assume that the car locations that maximize the probability above are unique ($C_{1i} \neq c_{1j}$ for all $i \neq j$).

Note: you don't need to produce a complicated proof for this question. It is acceptable to provide a clear explanation based on your intuitive understanding of the scenario.

(c) **[2 points (Written)]** For general K, what is the treewidth corresponding to the posterior distribution over all K car locations at all T time steps conditioned on all the sensor readings:

$$\mathbb{P}(C_{11} = c_{11}, \ldots, C_{1K} = c_{1K}, \ldots, C_{T1} = c_{T1}, \ldots, C_{TK} = c_{TK} \mid E_1 = e_1, \ldots, E_T = e_T)?$$

Briefly justify your answer.

This handout includes space for every question that requires a written response. Please feel free to use it to handwrite your solutions (legibly, please). If you choose to typeset your solutions, the `README.md` for this assignment includes instructions to regenerate this handout with your typeset LaTeX solutions.

1.a

1.b

1.c

5.a

5.b

5.c