CS 3600: Probabilistic Inference Project Hints

Osama Sakhi
Siyan Li

Georgia Institute of Technology

Contents

1 Notation:

Here we'll define the terms used within the project and how they may relate to concepts from the course.

1.1 `legalPositions`

These are the collection of coordinates $\mathcal{C}$ that are valid locations for Pacman or a ghost to be at. Let's call c any coordinate in `legalPositions`, i.e. $c \in \mathcal{C}$.

1.2 `beliefs`

If we let the random variable G represent ghost's location, then the `beliefs` represent the probability of finding a ghost at any given coordinate c on the grid. In other words:

$$P(G = c) \equiv \text{The probability a ghost can be found at position } c$$

1.3 `trueDistance`

The `trueDistance` is the distance from Pacman to some other coordinate c.

$$\texttt{trueDistance} \equiv \texttt{ManhattanDistance}(\text{Pacman}, c)$$

1.4 `noiseDistance`

The `noisyDistance` is the reading Pacman gets from using his sensor. It's supposed to give us a distance reading to the nearest ghost. This sensor isn't the best (i.e it's *noisy*), so we don't trust it to tell us the `trueDistance`, but there's probably some useful information the reading can give us. Let's denote observations with the random variable Y. When the agent makes a single observation, we'll denote that as $Y = y$, where y is the exact value just observed.

1.5 `emissionModel`

This model represents how much we believe that we should be observing the `noisyDistance` we just observed, *given* that we know the true distance to a ghost, `trueDistance`. In other words, we have a *conditional probability distribution* of the form:

$$P(Y|X) \equiv P(\texttt{noisyDistance}|\texttt{trueDistance})$$

In code, you will be given the `emissionModel` produced by the `noisyDistance` observation y. This `emissionModel` will come in the form of a Counter, with keys being the `trueDistance`s and values being the corresponding probabilities.

1.6 `jail position`

This is a special position on the grid, far, far away from Pacman. *Captured* ghosts end up here. No sensor can sense a ghost in this position. When a ghost has been *captured*, the belief that the ghost is anywhere but the jail cell must be 0%, and we must have 100% belief that the ghost is in the jail position. If a ghost is *not captured*, the belief that the ghost is in the jail position must be 0%.

Note: Only the ghost can ever be in the `jail position`.

2 Updating Beliefs with Observations

Let's say our agent has just started playing in our game world. He hasn't taken any observations with his sensor, and he currently believes that the ghost is equally likely to be anywhere. In other words:

$$P(G = c) = \alpha \text{ for every } c \in \mathcal{C}, \text{ where } \alpha = \frac{1}{\texttt{size}(\mathcal{C})} \text{ is some constant}$$

Now, let's say he gets one sensor reading. We can use that `noisyDistance` reading along with our `emissionModel` and our existing `beliefs` to determine a new set of `beliefs`. If we fix the previous `beliefs` table, we can iteratively build up a new one. This process can be repeated every time Pacman makes an observation.

Remember: A coordinate c is not the same as the `trueDistance` from that coordinate c to `Pacman`.

Hint: Think about how you can use product rule to capture the union of the ghost being at c, and the sensor reading a particular distance.

Hint: Multiple positions can have the same noisy distance, so it may be important to normalize your values. `util.Counters` has a convenient function for this.

3 Updating Beliefs with Elapsing Time

This form of updating beliefs differs from the previous in that Pacman is not making observations with his noisy sensor anymore. Instead, he has some model that tells him all the places that a ghost may move, *given* that the ghost was at some other known position. In other words, if Pacman assumed that a ghost was at some coordinate c at the last time step t, he can use `getPositionDistribution` to have a probability distribution of where that same ghost may be at time $(t + 1)$. If we let the random variable G_i denote the location of the ghost at time i, we have:

$$P(G_{t+1} = d | G_t = c) \equiv \text{The probability of the ghost transitioning from } c \text{ to } d$$

Remember, we don't actually know where the ghost is, so we can't assume $P(G_t = c) = 100\%$. Think about how we may be able to use what we already *believe* about the ghost's location to help us.

4 Hunting Multiple Ghosts

Our beloved Pacman is but one ghost hunter, so when he's tasked with hunting multiple ghosts, he's got to decide how he's going to go about capturing all of them. In other words, which actions he'll take. One easy way to go about chasing down multiple ghosts (especially in an environment where there isn't certainty about their locations) is to just chase the closest one first, then the next, and so on. This *greedy* approach involves three main steps:

1. Using the `beliefs` for each ghost, determine the single most probable location c that each ghost may be at. Treat these as the true locations of the ghosts.
2. Identify which of these ghost Pacman is already closest to.
3. Pick the action that brings Pacman as close to that ghost as possible.

5 Particle Filtering

In this part of the assignment, we'll switch it up a bit. Instead of using *probability distributions* as we had been using before, we will now use *particles* to represent the agent's understanding of where a ghost may be found.

5.1 Particles as *votes*

Particles are simply "votes" that represent the true *distribution* rather than the probabilities themselves. We can also convert between a *belief distribution* and *particles* at any time.

Imagine we have a bag of marbles of different colors:

- 95 blue
- 3 red
- 2 yellow

We can easily convert this to a distribution of colors:

- $P(\text{blue}) = 95\%$
- $P(\text{red}) = 3\%$
- $P(\text{yellow}) = 2\%$

The bag of marbles is analogous to the list of particles. Note how we can always go from a distribution to particles if we know how many particles we want total, which is always a given quantity.

Keep in mind that for each time step the most probable ghost position should have the highest number of particles "voting" for it. Some positions may get no votes at all.

5.2 Initializing Uniformly

Just like how we did in the *exact inference* sections, we want our agent to start the game believing that every position is equally likely to contain a ghost. For particle filtering, that means we should have an equal number of votes for each position.

6 Updating Particles with Observations

This is very similar to Section 2, in that we're using observations (`noisyDistance`) to update our `beliefs` of where the particles are. However, rather than a distribution called `beliefs`, we want to use a list of particles to represent the distribution. Think of how using particles may be different from having a prior explicitly of the form $P(C = c)$.

Hint: You'll want to build something that has a weight for every position the ghost may be in. This can then be normalized to form a distribution. Lastly, you can work backwards from a distribution to a list of particles by doing the reverse of what we did in Section 5.1.

Hint: There are lots of helper functions in `util.py` that can help with this portion. Take a look at `util.sample` and `util.nSample`.

7 Updating Particles with Time

This will be analogous to Section 3, but with the key difference being the use of particles again as opposed to a prior of the form $P(C = c)$. One speed optimization here will be to iterate over the key-value pairs of the `newPosDist` rather than going over all possible `legalPositions`, checking which ones have a weight in `newPosDist`, and then factoring that into your weighted representation before you convert that into your list of updated particles.

8 Joint Particle Filtering

Now our task has slightly changed. We're going to use Particle Filtering with multiple ghosts. These ghosts no longer behave independently of each other, so we cannot apply separate Particle Filters for each of these ghosts. Instead, we will now have particle tuples of the form:

$$(c_1, c_2, \ldots c_k)$$

Where k is the number of ghosts, and c_i represents some `legalPosition`, i.e $c_i \in \mathcal{C}$. There are *many* possible tuples since they are the length k *permutations* of the `legalPositions`, and the number of particles we consider may not be able to capture all of them.

8.1 Permutations

Suppose we have a 3×3 grid of legal positions, and we have $k = 3$ ghosts. The possible particle tuples are all permutations of length $k = 3$ from 9 possible cells, which is equal to $9^k = 9^3 = 729$.

8.2 Creating Particles

Now we know that the particle tuples are simply length-k permutations of the `legalPositions`. We can use `itertools.permutations` to create these permutations. Once we've done that, we may notice that the number of permutations may be greater than the number of particles we want, N. It would be naive to stick with the first N particle tuples since that may pick out only tuples placing each ghost in the top-right corner of the grid, for example. A smarter strategy would be to *randomly shuffle* the list of permutations, ensuring we're initializing particles randomly and not by some particular ordering.

8.3 Updating with Observations and Time

The mechanics of `observeStage` and `elapseTime` won't change much from our previous implementations. The key differences will be the need to do some additional book-keeping to account for the fact that our particles are now tuples representing each ghost rather than just a single ghost. Keep in mind that your implementation may need to resample for a single ghost (i.e. a single index in the tuples). Lastly, you'll need to account for some ghosts being in jails while the remaining ghosts run freely.