

Sucky Probabilistic Updates – Connecting the Equations with the Code

For this scenario, Sucky's "knowledge base" consists of a probability distribution over squares, where for example $P(G = (r, c))$ is the probability (belief) that the ghost is at location (r, c) . Or in other words, G is a random variable with a domain of size 100: $(1,1), (1,2), \dots, (10,10)$. I'll write $P(G = (r, c))$ as $P((r, c))$ just as shorthand, or even $P(s)$ where s is location, or "square."

In coding terms, (the `ghostprobs.py` module), we store these probabilities as an array or dictionary of 100 elements, and the sum of the values in the dictionary must sum to 1. (I'm going to just say dictionary from now on because that's how it is implemented in `ghostprobs.py`.) If Sucky wants to know the probability that the ghost is at a particular square, it just looks up the value of the square in the dictionary. If it wants to know the square with the highest probability of having the ghost, it iterates over the items in the dictionary, and chooses the index (square) with the highest value.

When Sucky does a *probe* action, the percept color it gets back will be one of $\{green, yellow, orange, red\}$, but also crucial is Sucky's location at the point where it did the probe. You'll notice that the world model gets the color as a percept, but it updates based on the percept $(color, currentLocation)$. That percept is sent to the **ghost_probs** module, which is responsible for recomputing the Ghost Probability for all 100 squares – the state of belief about *all* squares changes any time sensor information is received about *any* square.

Now let's focus on the probability update, starting with how to apply Bayes Rule. Suppose we get the percept $(color, currentLocation)$ and we need to update the probabilities to get from

$$P(otherLocation)$$

to

$$P(otherLocation \mid (color, currentLocation))$$

for all squares *otherLocation*. The updated probability is "the probability that the ghost is at square *otherLocation* given that I received a probe percept of color while located at *currentLocation*."

Applying Bayes Rule directly we get

$$\begin{aligned} &P(otherLocation \mid (color, currentLocation)) \\ &= \frac{P((color, currentLocation) \mid otherLocation)P(otherLocation)}{P((color, currentLocation))} \end{aligned}$$

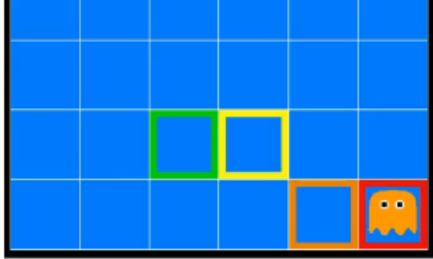
The three terms you need to compute to apply the rule are

1. $P((color, currentLocation) \mid otherLocation)$
2. $P(otherLocation)$
3. $P((color, currentLocation))$

The second one is easy – it’s the prior probability, and that is stored directly in the **ghostprobs** module.

For the first term, you need to review the material in the “Probability Fundamentals” video, right at the beginning. It talks about the “Noisy Sensor” model

- Sensor readings at a square tell how close a square is to the ghost
 - On the ghost: red
 - 1 or 2 away: orange
 - 3 or 4 away: yellow
 - 5+ away: green



▪ Sensors are noisy, but we know $P(\text{Color} \mid \text{Distance})$

$P(\text{red} \mid 3)$	$P(\text{orange} \mid 3)$	$P(\text{yellow} \mid 3)$	$P(\text{green} \mid 3)$
0.05	0.15	0.5	0.3

The image shows a video player interface with a 'Play' button, a progress bar at 1:12 / 49:00, and standard video controls (volume, settings, full screen).

And notice that the sensor behavior depends only on the distance between where the ghost is and where the probe was taken. You will also notice that the **ghostprobs** module stores those conditional probabilities in a variable **default_probe_probs**

These “noisy sensor probabilities” are exactly the conditional probabilities we are looking for. You saw those probabilities in the slides, and they are also stored in code in the variable `default_probe_probs`. For example, here are two lines from that variable, I arbitrarily chose the distances 0 and 3:

```
0: {"red": .94, "orange": .03, "yellow": .02, "green": .0}
3: {"red": .05, "orange": .15, "yellow": .5, "green": .3},
```

Which shows that if the probe square and the ghost square are the same (distance 0), the probe color is very likely to be red, but if they are 3 squares apart, the color is very unlikely to be red (but not impossible), and is much more likely to be yellow or green. Again, the algebraic equivalent of those dictionary entries are

$$P(\text{"red"} \mid \text{distance}(\text{currentLocation}, \text{otherLocation}) = 3) = .05$$

The code table/dictionary only goes up to a distance of 7, so assume that the probe sensor behaves the same for all distances ≥ 7 .

So you can get the first term in the Bayesian update in a single table lookup:

$$P((color, currentLocation) | otherLocation) \\ \equiv P(color | distance(currentLocation, otherLocation))$$

That leaves just the last term in the Bayes rule application, $P((color, currentLocation))$ which is the least intuitive. Until you realize that to get it, you need to iterate over all the squares, which is just saying you must consider the possibility that the ghost could be at any square. So we apply the rule of total probability and iterate over all squares s

$$P((color, currentLocation)) = \sum_s P((color, currentLocation | s)P(s))$$

But you already know how to compute both $P((color, currentLocation | s)$ and $P(s)$ for all squares s .

This is the calculation you will implement (for all squares) in the function **update_probe**. The result of **update_probe** will be to update all of the values in the variable **self.probs**, which is the variable containing the ghost probabilities for all squares.

(A common gotcha: you need to calculate the new probability for *all* squares before you change the value of `self.probs` for *any* square.)

The only other code you must write is for the function **update_bust**, which is called when Sucky does a bust action, which either comes back either True or False. The world model sends the probability updater a bust report like `(True, (r, c))`, which means the world says that the ghost is *definitely* at `(r, c)`. In that case you know what to do: you update `self.probs` to make the probability at `(r, c)` be `1.0`, and the probabilities at all other squares to be `0.0`. Notice that after doing so, the probabilities in `self.probs` still sum to `1.0`, which is a requirement.

Updating on a False bust report is a little more complicated: if the report is `(False, (r, c))` the agent now knows that the ghost is *not* at `(r, c)`, but otherwise has no additional information as to where the ghost is. So `self.probs[(r, c)]` gets set to `0.0`. But then you need to *normalize* the distribution: adjust the probabilities so they sum to `1.0` again. You do that by dividing each probability in `self.probs` with the sum:

$$\sum_{(r,c)} P(G = (r, c))$$

Complete Example (Non Code)

Here is a small example of doing the Bayesian update on a grid of size 2. I find it helpful to work out a small example by hand prior to coding. Plus you can use the hand calculations to test your **ghostprobs** code (notice in the **ghostprobs** constructor you can supply a different grid size, and also different probe probabilities).

I will use these probe probabilities for the example

Distance	Color			
	<i>red</i>	<i>orange</i>	<i>yellow</i>	<i>green</i>
0	.95	.03	.01	.01
1	.20	.95	.01	.01
2	.05	.90	.03	.02

These numbers are made up, they are just supposed to reflect the example in the video, that *red* means “on the ghost,” *orange* means “1 or 2 away”, etc., except the probes are sometimes wrong.

Verify your understanding that the table entry of .90 under distance = 2 and color = orange means $P(\text{“orange”} \mid \text{distance} = 2) = .90$

First we assume a uniform initial prior:

$$P((1,1)) = P((1,2)) = P((2,1)) = P((2,2)) = 0.25$$

Let’s calculate the updated ghost probabilities based on a probe reading of *orange* that was taken at (2,2).

First we’ll calculate $P((1,1) \mid \text{orange at } (2,2))$

$$P((1,1) \mid \text{orange at } (2,2)) = \frac{P(\text{orange at } (2,2) \mid (1,1)) P((1,1))}{P(\text{orange at } (2,2))}$$

First term: $P(\text{orange at } (2,2) \mid (1,1))$ is equivalent to saying $P(\text{orange} \mid \text{distance} = 2)$ because the only thing affecting the probe is the distance between the probed square and the ghost square. We read that value of .90 from the table.

Second term: $P((1,1))$ is the prior, in this case 0.25.

Third term: we apply the law of total probability and sum over all possible squares. Because we know there is exactly one ghost and it has to be in one of the squares, $\sum_s P(s) = 1$

$$P(\text{orange at } (2,2)) = \sum_s P(\text{orange at } (2,2) \mid s) P(s)$$

$$P(\text{orange at } (2,2)) = \sum_s P(\text{orange} | \text{distance}(s, (2,2))) P(s)$$

$$= P(\text{orange} | \text{distance}((1,1), (2,2))) P((1,1)) + \\ P(\text{orange} | \text{distance}((1,2), (2,2))) P((1,2)) + \\ P(\text{orange} | \text{distance}((2,1), (2,2))) P((2,1)) + \\ P(\text{orange} | \text{distance}((2,2), (2,2))) P((2,2))$$

$$P(\text{orange at } (2,2))$$

$$= P(\text{orange} | 2) * 0.25 + P(\text{orange} | 1) * 0.25 + P(\text{orange} | 1) * 0.25 \\ + P(\text{orange} | 0) * 0.25$$

$$P(\text{orange at } (2,2)) = (.90 + .95 + .95 + .03) * 0.25$$

$$P(\text{orange at } (2,2)) = .7075$$

And now the updated ghost probability for (1,1) after the probe

$$P((1,1) | \text{orange at } (2,2)) = \frac{.90 * .25}{.7075} = .318021$$

Calculating the updated probability for the other squares is easier, for example the probability that the ghost is at (2,2) should go down after getting the *orange* probe at (2,2) because according to the table, it is unlikely that the probe will be *orange* if the probe is in the same square as the ghost.

$$P((2,2) | \text{orange at } (2,2)) = \frac{P(\text{orange} | \text{distance} = 0) P((2,2))}{P(\text{orange at } (2,2))} = \frac{.03 * .25}{.7075} = .010601$$

With this example I would suggest

1. Try doing the example on paper without looking at the solution, for all four squares
2. Once you understand the calculation steps, do the implementation in the **ghostprobs** module
3. Write some code to run the **ghostprobs** on this small example, and verify that your module is calculating correctly when you do an **update_probe('orange', (2,2))**
4. You can then go on to implement **update_bust**, then try the bigger test in **test_ghost_probs()**