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 $P\big(G=(r,c)\big)$ 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), …(10,10). I'll write $P\big(G=(r,c)\big)$ as $P\big((r,c)\big)$ 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

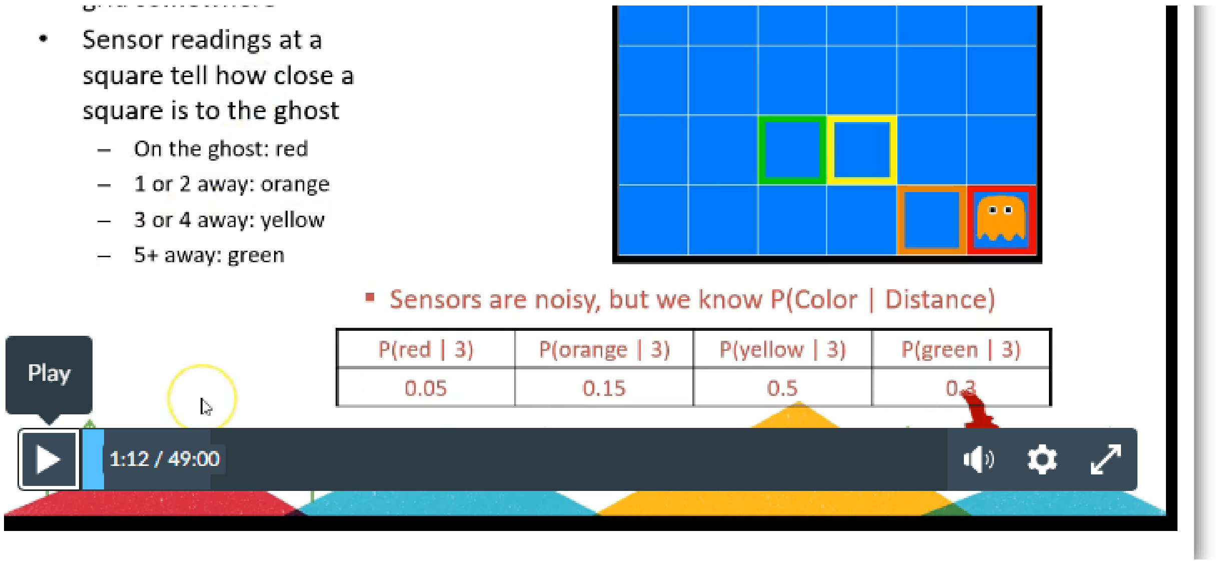
$$P(otherLocation|(color, currentLocation)) = \frac{P((color, currentLocation)|otherLocation)P(otherLocation)}{P((color,\ currentLocation))}$$

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

1. $P\big((color, currentLocation)\big|otherLocation\big)$
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(Color | Distance)

P(red | 3)	P(orange | 3)	P(yellow | 3)	P(green | 3)
0.05	0.15	0.5	0.3

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

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

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:

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

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*

$$P((color, currentLocation)) = \sum_{s} P((color, currentLocation \mid s)P(s)$$

But you already know how to compute both $P((color, currentLocation \mid s)$ 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:

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