Adam's Notes and Hints!

**Q 0.**

**Implement:**

inference.py: DiscreteDistribution.normalize()

inference.py: DiscreteDistribution.sample()

**Test:**

python -m doctest -v inference.py

DiscreteDistribution extends a Dictionary to be a discrete distribution

What's a discrete distribution?

A set of key / value pairs

Keys are the "event"

Values are "liklihoods" (Probabilities between 0 .. 1) that that event happened.

- The sum of the probabilities for all keys should add up to 1.

Functions to implement:

- normalize: Force the sum of all probabilities to add up to one. Pretty easy.

- Two edge cases - make sure the dictionary is not empty, and all values are not zero

- otherwise, divide all values by the total

Sample: Slightly more tricky, we need to return an event randomly based on the distribution

\* Check if not normalized, and if not, normalize - this will make some sections less difficult

Eg. imagine we had 4 events ('a', 'b', 'c', 'd') with probabilities:

'a' : 0.1

'b' : 0.2

'c' : 0.3

'd' : 0.4

Visualizing our distribution

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

a b c d

If we partition out the space for these events from 0...1

0 0.1 0.3 0.6 1

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

a b c d

Given a random number, eg. random.random() => 0...0.999

- Whichever range that number falls in, we need to return that number

- eg. if random.random() returned 0.05 -> This is in the 'a' range, return 'a'

- eg. if random.random() returned 0.4 -> This is in the 'c' range, return 'c'

- eg. if random.random() returned 0.743 -> This is in the 'd' range, return 'd'

Put another way, if we sampled 10,000 numbers

~ 1,000 would be in a

~ 2,000 would be in b

~ 3,000 would be in c

~ 4,000 would be in d

This question is worth zero marks, but it really helps you understand that data structure you will use.

No autograder for this question, you can verify correct operation with doc-tests

python -m doctest -v inference.py

This is the code in the docstring, first thing in the function:

    def sample(self):

        """

        Draw a random sample from the distribution and return the key, weighted

        by the values associated with each key.

        >>> dist = DiscreteDistribution()

        >>> dist['a'] = 1

        >>> dist['b'] = 2

        >>> dist['c'] = 3

        >>> dist['d'] = 4

        >>> N = 100000.0

        >>> samples = [dist.sample() for \_ in range(int(N))]

        >>> round(samples.count('a') \* 1.0/N, 1)  # proportion of 'a'

        0.1

        >>> round(samples.count('b') \* 1.0/N, 1)

        0.2

        >>> round(samples.count('c') \* 1.0/N, 1)

        0.3

        >>> round(samples.count('d') \* 1.0/N, 1)

        0.4

        """

This class extends dictionary. That means, to use the underlying dictionary structure, you can use the "self" function inside of it, eg.

self['a'] = 3

You can check if the dictionary is empty by using self as a bool

You can iterate over the dictionary using "for key in self:" then "self[key]"

**Q 1. Part 1: HMMs**

**Implement:**

inference.py: InferenceModule.getObeservationProb( ... )

**Test:**

python autograder.py -q q1

Given a noisy observation, what is the likelihood that event occurred?

Passed into the function:

noisyDistance - distance computed to the noisy reading

pacmanPosition - pacman's position (actual)

ghostPosition - ghost position (actual)

jailPosition - where we store the ghosts after they have been captured

we can compute the true distance between pacman and the ghost using manhattan distance

This function will give you the observation probability

- busters.getObservationProbability(noisyDistance, trueDistance)

- However, 3 edge cases to handle:

- ghost is in jail and we are probing jail location -> return 1.0

- ghost is in jail and we are probing non-jail location -> return 0.0

- ghost is not in jail and we are probing jail location -> return 0.0

- ghost is not in jail and we are probiling a non-jail location - defer to the function given.

**Q 2.**

**Implement:**

inference.py: ExactInference.observeUpdate( ... )

**Test:**

python autograder.py -q q2 --no-graphics

Recall in HMM's there are two steps - observing updates (getting evidence which sharpens beliefs) and time passing (beliefs dull)

Things to access:

self.beliefs - a DiscreteDistribution of our beliefs, eg. how likely a ghost is in a location

gameState.getPacmanPosition()

self.getJailPosition()

self.allPositions - a list of all legal positions ghosts may be, including jail

self.getObservationProb - you coded this in (1)

Our goal: Given an observation, for each position, update your belief based on the probability that the ghost is in that position

- also recall that a belief update function is: P(Xt) \* B'(Xt-1)

\* don't forget to normalize your beliefs when done!

**Q 3.**

**Implement:**

inference.py: ExactInference.elapseTime( ... )

**Test:**

python autograder.py -q q3 --no-graphics

The other thing that can happen in HMM's - Time passes. Ghosts move. We become less certain about where they are.

Given a position, we need to probe the game to find a new distribution for what is likely to happen in that location

This one is kind of tricky - let's do a worked example

Example: Ghosts in a tiny space with random movement:

In code: You can get the distribution of where they will go based on where they are using:

newPosDist = self.getPositionDistribution(gameState, oldPos)

Beyond this it's just looping through dictionaries, taking products and sums as needed.

**Q 4.**

**Implement:**

bustersAgents.py: GreedyBustersAgent.chooseAction( ... )

**Test:**

python autograder.py -q q4 --no-graphics

We want to code up a greedy pacman which:

- For a list of distributions of ghost locations (livingGhostPositionDistributions)

- Finds the most likely position estimate of the ghost (ie. highest probability)

- For the list of estimated ghost positions

- Find the closest ghost using the distance metric (self.distancer.getDistance( ... ) )

- For the available actions

- Take the one which moves us closest to the ghost

**Q5: Part 2: Particle Filtering**

**Implement:**

inference.py: ParticleFilter.initializeUniformly( ... )

inference.py: ParticleFilter.getBeliefDistribution( ... )

**Test:**

python autograder.py -q q5

Recall: Particle Filtering (Quick Example)

Data Structure:

self.particles: a list of particles positions

initializeUniformely - we want to uniformly distribute all of our particles across legal positions

getBeliefDistribution - we basically want to turn our particle list into a normalized distribution

- ie. add each particle to the dictionary with a weight of 1

- then normalize

\* You may need to come back to this question if your Q8 solution isn't working!

**Q6:**

**Implement:**

inference.py: ParticleFilter.observeUpdate( ... )

**Test:**

python autograder.py -q q6 --no-graphics

As before, we are getting some new evidence, but now we want to update our particles accordingly

- first of all, we need to weight each of our given particles

- next we need to make a distribution for each legal particle using the sum of the assigned weights

- finally we need to resample from this distribution for each of our particles - throw out the old particle locations and make new ones!

- Let's do an example

\* Handle the edge case when all weights are zero - then we just need to uniformely initialize

**Q7:**

**Implement:**

inference.py: ParticleFilter.elapseTime( ... )

**Test:**

python autograder.py -q q7 --no-graphics

This is where the strength of the particle filter really becomes apparent - if we think of each particle as a potential ghost with a given probability, to figure out where it would go with time, we just need to simulate movement for that ghost as a probability distribution, and then sample from that distribution

Recall: We simulate movement using:

newPosDist = self.getPositionDistribution(gameState, oldPos)

This gives us a distribution of possible locations. To set our new particle location, we just need to sample from this distribution

**Q8: Part 3: Joint Distribution Particle Filtering**

**Implement:**

inference.py: JointParticleFilter.initializeUniformly( ... )

\* Ensure that ParticleFilter.getBeliefDistribution works with particles of multiple positions!

**Test:**

python autograder.py -q q8 --no-graphics

self.legalPositions - all of the legal ghost positions

self.numGhosts - the number of ghosts

self.numParticles - the number of particles

We now want to generalize our Particle Filtering to work for multiple ghosts. Because ghosts factor into other ghosts movement, this is a Bayes Net. However, recall we can also think of these Bayes Nets as HMMs, where the states and evidence are actually combined from several states and several observations.

For the first question, we just need to initialize our particles uniformly.

What does a particle look like now? Tuples

2 ghosts: ( (1, 0), (3, 5) )

3 ghosts: ( (1, 0), (3, 5), (6, 8) )

So, given a list of legal positions, eg. [ (1, 0), (3, 5), (6, 8) ], we need to create a list of particle positions that are combinations of these, eg for two ghosts:

[ ((1,0), (3,5)), ((1,0),(6,8)), ((3,5),(6,8)) ]

And then uniformly distribute our particles across these states.

We can do this with python itertools product, but be sure to shuffle the results after!

\* be sure to code generally. Account for 2 ghosts, 3 ghosts, ... n ghosts.

**Q9:**

**Implement:**

inference.py: JointParticleFilter.observeUpdate( ... )

**Test:**

python autograder.py -q q9 --no-graphics

ObserveUpdate for JointDistributions is an extension of single particles

- The difference is, we now need to weight our particles by the product of the probabilities of all the ghosts observations

- ie. for a given particle ( tuple of ghost position) , we need to get the observation probability that each ghost is in their corresponding location, and multiply these probabilities together

- Again, we need to think generally. How can we write this to work for 'n' ghosts?

- P(x1) \* P(x2) \* .... \* P (xn) <-- how would you put this into a loop?

- Also, don't forget that if a particle appeared more than once in our original list of particles, when building the distribution we need to sum the weights for that particle.

Once we have built our weight distribution, we again throw out all of our old particles, and resample from it to get all of our new particles

**Q9:**

**Implement:**

inference.py: JointParticleFilter.elapseTime( ... )

**Test:**

python autograder.py -q q10 --no-graphics

As before, the other part of our system is, when time passes, we need to simulate the movement for the ghost

- The difference is now we need to do it for a tuple of n ghosts

- So, for each ghost - get the appropriate position distribution

- As there are multiple ghosts, we have a new way to access this

- self.getPositionDistribution(gameState, particle\_as\_a\_list, current\_ghost\_index, ghost\_agent)

- Finally, we just need to sample from this distribution for each ghost in our particle

- Recall tuples are immutable, so the code casts a tuple to a list, lets you make the changes as needed, and then casts it back to a tuple at the end.