Project 4a

Introduction

Project 4a will be focusing on inference, using Bayesian networks and Particle Filtering. First you will be implementing a parser for a Bayesian network that calculates probabilities of assumptions given observations. Next you will use a dynamic Bayes' Net to help pacman track ghosts using particle filtering.

You will be modifying the files `bayesNet.py` and `inference.py`.

Questions

Question 0 (0 points): DiscreteDistribution Class

Throughout this project, we will be using the *DiscreteDistribution* class defined in *inference.py* to model belief distributions and weight distributions. This class is an extension of the built-in Python *dictionary* class, where the keys are the different discrete elements of our distribution, and the corresponding values are proportional to the belief or weight that the distribution assigns that element. This question asks you to fill in the missing parts of this class, which will be crucial for later questions (even though this question itself is worth no points).

First, fill in the *normalize* method, which normalizes the values in the distribution to sum to one, but keeps the proportions of the values the same. Use the *total* method to find the sum of the values in the distribution. For an empty distribution or a distribution where all of the values are zero, do nothing. Note that this method modifies the distribution directly, rather than returning a new distribution.

Second, fill in the *sample* method, which draws a sample from the distribution, where the probability that a key is sampled is proportional to its corresponding value. Assume that the distribution is not empty, and not all of the values are zero. Note that the distribution does not necessarily have to be normalized prior to calling this method. You may find Python's built-in *random.random()* function useful for this question.

There are no autograder tests for this question, but the correctness of your implementation can be easily checked by looking at the values after the function is called.

Question 1 (2 points): Observation Probability

In this question, you will implement the *getObservationProb* method in the *InferenceModule* base class in *inference.py*. This method takes in an *observation* (which is a noisy reading of the distance to the ghost), Pacman's position, the ghost's position, and the position of the ghost's jail, and returns the probability of the noisy distance reading given Pacman's position and the ghost's position. In other words, we want to return *P(noisyAction | pacmanPosition, ghostPosition).*

The distance sensor has a probability distribution over distance readings given the true distance from Pacman to the ghost. This distribution is modeled by the function busters.getObservationProbability(noisyAction, trueDistance), which returns P(noisyAction | oldGhostPos, newGhostPos) and is provided for you. You should use this function to help you solve the problem.

To test your code and run the autograder for this question:

```
python autograder.py -q q1
```

As a general note, it is possible for some of the autograder tests to take a long time to run for this project, and you will have to exercise patience. As long as the autograder doesn't time out, you should be fine (provided that you actually pass the tests).

Question 2 (3 point) - Bayes' Theorem for Single Observation

For this section of the project we will be modifying the function *singleInference* in the BayesNetwork class of bayesNet.py*.*

In the **Russel and Norvig** text on page 512 there is an example of a Bayes Network.

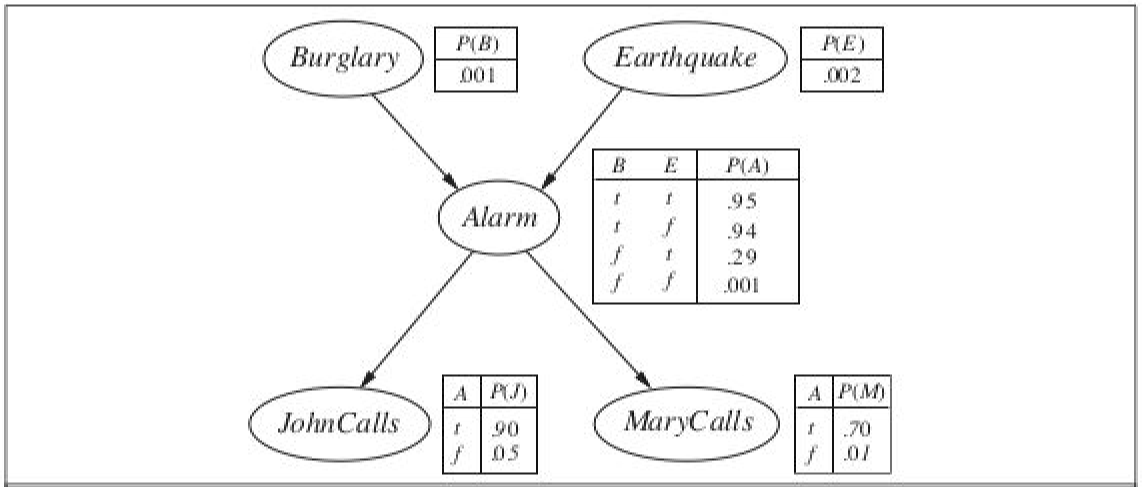

Figure 14.2 A typical Bayesian network, showing both the topology and the conditional probability tables (CPTs). In the CPTs, the letters B, E, A, J, and M stand for *Burglary*, *Earthquake*, *Alarm*, *JohnCalls*, and *MaryCalls*, respectively.

In a Bayes Network edges between nodes are directed and each node has a conditional probability table (CPT). In the constructor for the BayesNetwork class you will see that a network object is passed into the class.

Please make sure to read the documentation for the CPT class in *bayesNet.py*

The function *singleInference* is given two nodes, A and B. Using the Bayes Network, or whatever other structures you decide to create, return p(A|B), that is to say the probability of A given B.

To calculate this you will be using Bayes' Theorem, which looks like

$$P(\mathrm{A}|\mathrm{B}) = \frac{P(\mathrm{B}|A)\cdot P(A)}{P(B)}$$

Note*:* There are many ways to approach this question, and you aren't necessarily restricted to just the network structure. Feel free to think what other structures may be useful in solving this question, and add them to the constructor for your use.

You can test you solution to question 2 by running

```
python autograder.py -q q2
```

Question 3 (4 points) - Bayes' Theorem for Multiple Observations

For this section of the project we will be modifying the function *multipleInference* in the *BayesNetwork* class of *BayesNetwork.py.*

In the previous question you were only given one observation, B, and told to calculate the probability of A given B. For question 3 you will instead be given. A list named *observations* which contains any number of nodes from the Bayes net. Using this, calculate the probability of A.

Note: In this case you are calculating p(A|B) where p(B) is equal to the probability of every node in *observations* anded. That is to say, if *observations* contains nodes C and D, then the p(B) in this case = p(C and D).

Question 4 (2 points): Approximate Inference Initialization and Beliefs

Approximate inference is very trendy among ghost hunters this season. For the next few questions, you will implement a particle filtering algorithm for tracking a single ghost.

First, implement the functions *initializeUniformly* and *getBeliefDistribution* in the *ParticleFilter* class in *inference.py*. A particle (sample) is a ghost position in this inference problem. Note that, for initialization, particles should be evenly (not randomly) distributed across legal positions in order to ensure a uniform prior.

Note that the variable you store your particles in must be a list. A list is simply a collection of unweighted variables (positions in this case). Storing your particles as any other data type, such as a dictionary, is incorrect and will produce errors. The *getBeliefDistribution* method then takes the list of particles and converts it into a *DiscreteDistribution* object.

To test your code and run the autograder for this question:

```
python autograder.py -q q4
```

Question 5 (3 points): Approximate Inference Observation

Next, we will implement the *observeUpdate* method in the *ParticleFilter* class in *inference.py*. This method constructs a weight distribution over *self.particles* where the weight of a particle is the probability of the observation given Pacman's position and that particle location. Then, we resample from this weighted distribution to construct our new list of particles.

You should again use the function *self.getObservationProb* to find the probability of an observation given Pacman's position, a potential ghost position, and the jail position. The sample method of the *DiscreteDistribution* class will also be useful. As a reminder, you can obtain Pacman's position using *gameState.getPacmanPosition(),* and the jail position using *self.getJailPosition().*

There is one special case that a correct implementation must handle. When all particles receive zero weight, the list of particles should be reinitialized by calling initializeUniformly. The total method of the DiscreteDistribution may be useful.

To run the autograder for this question and visualize the output:

```
python autograder.py -q q5
```

If you want to run this test (or any of the other tests) without graphics you can add the following flag:

```
python autograder.py -q q5 --no-graphics
```

IMPORTANT: In general, it is possible sometimes for the autograder to time out if running the tests with graphics. To accurately determine whether or not your code is efficient enough, you should run the tests with the --no-graphics flag. If the autograder passes with this flag, then you will receive full points, even if the autograder times out with graphics.

Submission

Please submit your assignment through Gradescope, submitting `bayesNet.py` and `inference.py`