Project 4b

Introduction

Project 4b provides a continuation of project 4a, where you will be implementing Exact Inference and Particle Filtering. Please note that for portions of the code of 4b to run you will have to have completed the sections from 4a in *inference.py*. The functions will be left blank in 4b's *inference.py* so you can copy them over from 4a.

You will be modifying the files *bustersAgents.py* and *inference.py.*

Questions

Question 1 (2 points): Observation Probability

(3 points): Exact Inference Observation

In this question, you will implement the *observeUpdate* method in *ExactInference* class of *inference.py* to correctly update the agent's belief distribution over ghost positions given an observation from Pacman's sensors. You are implementing the online belief update for observing new evidence. The *observeUpdate* method should, for this problem, update the belief at every position on the map after receiving a sensor reading. You should iterate your updates over the variable *self.allPositions* which includes all legal positions plus the special jail position. Beliefs represent the probability that the ghost is at a particular location, and are stored as a *DiscreteDistribution* object in a field called *self.beliefs*, which you should update.

Before typing any code, write down the equation of the inference problem you are trying to solve. You should use the function *self.getObservationProb* that you wrote in the last 4a, which returns the probability of an observation given an old ghost position, a new ghost position and the jail position. You can obtain Pacman's position using *gameState.getPacmanPosition(),* and the jail position using *self.getJailPosition().*

In the Pacman display, high posterior beliefs are represented by bright colors, while low beliefs are represented by dim colors. You should start with a large cloud of belief that shrinks over time as more evidence accumulates. As you watch the test cases, be sure that you understand how the squares converge to their final coloring.

Note: your busters agents have a separate inference module for each ghost they are tracking. That's why if you print an observation inside the *observeUpdate* function, you'll only see a single number even though there may be multiple ghosts on the board.

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

```
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 points): Exact Inference with Time Elapse

In the previous question you implemented belief updates for Pacman based on his observations. Fortunately, Pacman's observations are not his only source of knowledge about where a ghost may be. Pacman also has knowledge about the ways that a ghost may move; namely that the ghost can not move through a wall or more than one space in one time step.

To understand why this is useful to Pacman, consider the following scenario in which there is Pacman and one Ghost. Pacman receives many observations which indicate the ghost is very near, but then one which indicates the ghost is very far. The reading indicating the ghost is very far is likely to be the result of a buggy sensor. Pacman's prior knowledge of how the ghost may move will decrease the impact of this reading since Pacman knows the ghost could not move so far in only one move.

In this question, you will implement the *elapseTime* method in *ExactInference*. The *elapseTime* step should, for this problem, update the belief at every position on the map after one time step elapsing. Your agent has access to the action distribution for the ghost through *self.getPositionDistribution*. In order to obtain the distribution over new positions for the ghost, given its previous position, use this line of code:

```
newPosDist = self.getPositionDistribution(gameState, oldPos)
```

Where *oldPos* refers to the previous ghost position. *newPosDist* is a *DiscreteDistribution* object, where for each *position p* in self.*allPositions*, *newPosDist[p]* is the probability that the ghost is at *position p* at *time t + 1*, given that the ghost is at position *oldPos* at *time t.* Note that this call can be fairly expensive, so if your code is timing out, one thing to think about is whether or not you can reduce the number of calls to *self.getPositionDistribution*.

Before typing any code, write down the equation of the inference problem you are trying to solve. In order to test your predict implementation separately from your update implementation in the previous question, this question will not make use of your update implementation.

Since Pacman is not observing the ghost, this means the ghost's actions will not impact Pacman's beliefs. Over time, Pacman's beliefs will come to reflect places on the board where he believes ghosts are most likely to be given the geometry of the board and what Pacman already knows about their valid movements.

For the tests in this question we will sometimes use a ghost with random movements and other times we will use the *GoSouthGhost*. This ghost tends to move south so over time, and without any observations, Pacman's belief distribution should begin to focus around the bottom of the board. To see which ghost is used for each test case you can look in the .test files.

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

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

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 q2 --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.

As you watch the autograder output, remember that lighter squares indicate that pacman believes a ghost is more likely to occupy that location, and darker squares indicate a ghost is less likely to occupy that location. For which of the test cases do you notice differences emerging in the shading of the squares? Can you explain why some squares get lighter and some squares get darker?

Question 3 (2 points): Exact Inference Full Test

Now that Pacman knows how to use both his prior knowledge and his observations when figuring out where a ghost is, he is ready to hunt down ghosts on his own. This question will use your *observeUpdate* and *elapseTime* implementations together, along with a simple greedy hunting strategy which you will implement for this question. In the simple greedy strategy, Pacman assumes that each ghost is in its most likely position according to his beliefs, then moves toward the closest ghost. Up to this point, Pacman has moved by randomly selecting a valid action.

Implement the *chooseAction* method in *GreedyBustersAgent* in *bustersAgents.py*. Your agent should first find the most likely position of each remaining uncaptured ghost, then choose an action that minimizes the maze distance to the closest ghost.

To find the maze distance between any two positions *pos1* and *pos2*, use *self.distancer.getDistance(pos1, pos2).* To find the successor position of a position after an action:

```
successorPosition = Actions.getSuccessor(position, action)
```

You are provided with *livingGhostPositionDistributions*, a list of *DiscreteDistribution* objects representing the position belief distributions for each of the ghosts that are still uncaptured.

If correctly implemented, your agent should win the game in q4/3-gameScoreTest with a score greater than 700 at least 8 out of 10 times. Note: the autograder will also check the correctness of your inference directly, but the outcome of games is a reasonable sanity check.

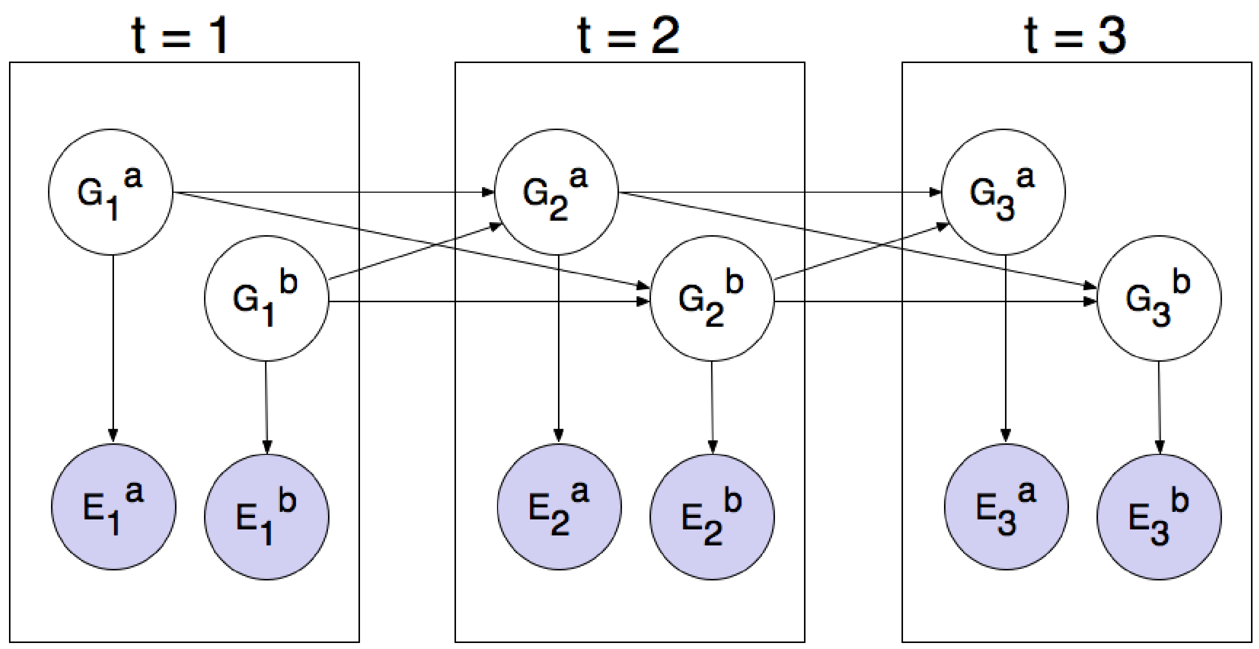
To run the autograder for this question and visualize the output:

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

Question 4 (1 points): Joint Particle Filter Observation

So far, we have tracked each ghost independently, which works fine for the default *RandomGhost* or more advanced *DirectionalGhost*. However, the prized *DispersingGhost* chooses actions that avoid other ghosts. Since the ghosts' transition models are no longer independent, all ghosts must be tracked jointly in a dynamic Bayes net!

The Bayes net has the following structure, where the hidden variables *G* represent ghost positions and the emission variables *E* are the noisy observations of each ghost's action, with previous position being stored at the previous time step. This structure can be extended to more ghosts, but only two (a and b) are shown below.



You will now implement a particle filter that tracks multiple ghosts simultaneously. Each particle will represent a tuple of ghost positions that is a sample of where all the ghosts are at the present time. The code is already set up to extract marginal distributions about each ghost from the joint inference algorithm you will create, so that belief clouds about individual ghosts can be displayed.

Complete the *initializeUniformly* method in *JointParticleFilter* in *inference.py*. Your initialization should be consistent with a uniform prior. You may find the Python itertools package helpful. Specifically, look at itertools.product to get an implementation of the Cartesian product. However, note that, if you use this, the permutations are not returned in a random order. Therefore, you must then shuffle the list of permutations in order to ensure even placement of particles across the board.

As before, use *self.legalPositions* to obtain a list of positions a ghost may occupy. Also as before, **the variable you store your particles in must be a list**.

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

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

Question 5 (3 points): Joint Particle Filter Observation

In this question, you will complete the *observeUpdate* method in the *JointParticleFilter* class of *inference.py*. A correct implementation will weight and resample the entire list of particles based on the observation of all ghost distances.

To loop over all the ghosts, use:

```
for i in range(self.numGhosts):
    ...
```

You can still obtain Pacman's position using *gameState.getPacmanPosition*(), but to get the jail position for a ghost, use *self.getJailPosition(i),* since now there are multiple ghosts each with their own jail positions.

Your implementation should also again handle the special case when all particles receive zero weight. In this case, self.particles should be recreated from the prior distribution by calling initializeUniformly.

As in the update method for the *ParticleFilter* class, 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.

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

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

Submission

Please submit your assignment through Gradescope, submitting *bustersAgents.py* and *inference.py*